UNIVERSITÀ DEGLI STUDI DI PISA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

PROGETTAZIONE E SVILUPPO DI UNO STRUMENTO PER LA TUTELA DELLA RISERVATEZZA DELLE INFORMAZIONI IN APPLICAZIONI .NET

Relatore: Prof.ssa C. Bernardeschi

Correlatore: Prof.ssa N. DeFrancesco

Candidato: Nicola Alessandro Provenzano

ANNO ACCADEMICO 2002–2003

Ai miei genitori

Indice

Elenco delle figure

Riassunto analitico

In questa tesi si propone uno strumento di analisi statica del codice atto a rilevare il trattamento improprio di informazioni personali da parte di applicazioni .NET.

Il flusso di propagazione delle informazioni viene tracciato in base ad impostazioni di livello di sicurezza, fornite in modo manuale o automatico ai Tipi definiti nell'applicazione .NET.

A partire dalle informazioni di sicurezza vengono definite una serie di regole atte a formalizzare il comportamento del codice .NET in questo nuovo contesto.

Obiettivo finale del lavoro è stato lo sviluppo di uno strumento in grado di rilevare possibili violazioni della privacy, dovute all'esecuzione di programmi che hanno accesso ad informazioni riservate.

Prefazione

Microsoft .NET è un insieme di tecnologie concepite per garantire l'utilizzo e l'interconnessione di sistemi software in totale sicurezza. Il meccanismo di "Code Access Security" e "Role Based Security", che garantiscono un sistema di protezione basato sulla tipologia e provenienza del codice, fornisce un efficace meccanismo di protezione, che arricchisce, in certi casi implementa (Windows 98), le politiche di sicurezza del Sistema Operativo.

Entrambi i meccanismi non controllano la propagazione delle informazioni a cui è stato concesso l'accesso: informazioni riservate possono essere memorizzate dall'applicazione, per esempio, in risorse di dominio pubblico.

Questo problema, noto in letteratura come "Secure Information Flow" [14], è stato affrontato per l'architettura Java in [1].

Questa tesi propone i risultati ottenuti dallo studio dell'architettura .NET ed espone le problematiche connesse con il Secure Information Flow, di cui il codice .NET non è esente. Si descrive, infine, l'implementazione ed il funzionamento di un Tool di analisi statica, appositamente progettato e sviluppato per individuare violazioni al trattamento improprio di informazioni personali in .NET.

Introduzione

La grande diffusione della banda larga, la maggiore disponibilità di risorse di calcolo e la grande richiesta di applicazioni di intrattenimento hanno modificato le attuali tendenze nel settore del software che si sono orientate alla promozione di applicazioni lato client in grado di offrire i più svariati servizi.

In questo contesto diventa di primaria importanza il controllo dell'affidabilità del codice dell'applicazione e delle risorse cui potrebbe accedere.

Le architetture di protezione tradizionali si basano sugli account utente per il controllo degli accessi e la definizione del livello di isolamento dei sistemi. Entro questi limiti, al codice vengono comunque assegnati diritti di accesso completi sulla base del presupposto che tutto il codice eseguito da un particolare utente è da considerarsi ugualmente affidabile. Sfortunatamente, l'isolamento del codice in base agli utenti non è sufficiente per proteggere un programma da un altro, in particolare se entrambi i programmi vengono eseguiti nel contesto di sicurezza dello stesso utente.

Un modello di protezione alternativo, prevede, invece, che il codice, considerato non totalmente affidabile, venga relegato a un modello di esecuzione "sandbox", in base al quale il codice viene eseguito in un ambiente isolato senza poter accedere alla maggior parte dei servizi.

Microsoft .NET è un insieme di tecnologie che trovano un supporto per lo sviluppo nel FrameWork .NET.

Le applicazioni sviluppate utilizzando il FrameWork .NET sono distribuite in un linguaggio intermedio orientato agli oggetti chiamato Common Intermediate Language (CIL) o, anche, Microsoft Intermediate Language (MSIL) ed eseguite da un ambiente di esecuzione, praticamente una virtual machine, chiamato Common Language Runtime (CLR).

Microsoft .NET propone una soluzione al problema della sicurezza che trova un giusto equilibrio tra i due meccanismi di protezione precedenti.

Comunque entrambi i meccanismi non controllano la propagazione delle informazioni a cui è stato concesso l'accesso: informazioni riservate possono essere memorizzate dall'applicazione, per esempio, in risorse di dominio pubblico.

In questa tesi viene analizzato il problema della tutela della riservatezza delle informazioni private, noto come flusso di informazione sicuro (SIF) in applicazioni .NET. Per individuare la presenza di flussi illeciti di informazioni viene utilizzato un approccio per la verifica della sicurezza delle informazioni sviluppato in [1] e rielaborato per tenere conto delle differenze introdotte dal nuovo ambiente. Si propone, inoltre, un Tool, chiamato SIFDotNet, appositamente progettato e sviluppato per l'analisi del Secure Information Flow nel Common Intermediate Language.

SIFDotNet è grado di disassemblare applicazioni .NET, di gestire l'input di impostazioni di sicurezza, comunicando in XML, e di testare il corretto rispetto della privacy per un sottoinsieme significativo delle istruzioni CIL.

La tesi è così organizzata:

- Capitolo 1 - Il Framework .NET: Viene descritta l'architettura .NET.
- Capitolo 2 - All'interno del CLR: Panoramica sulle tecnologie alla base del Common Language Runtime.
- Capitolo 3 - Microsoft .NET sicurezza: Panoramica sulla sicurezza in ambiente .NET.
- Capitolo 4 - I limiti della sicurezza in .NET
- Capitolo 5 - Flusso sicuro delle informazioni
- Capitolo 6 - SIFDotNet: un tool per la tutela delle informazioni private
- Capitolo 7 - Applicazione di SIFDotNet ad un caso reale
- Capitolo 8 - Conclusioni

Parte I

Microsoft .NET

Capitolo 1

Il Framework .NET

1.1 Che cos'è il Framework .NET?

Il termine .NET Framework si riferisce al gruppo di tecnologie che costituiscono la base di sviluppo della piattaforma Microsoft .NET [2]. Le principali tecnologie di questo gruppo sono rappresentate dal run-time e dalla libreria di classi (BCL Base Class Library), come illustrato nella Figura 1.1.

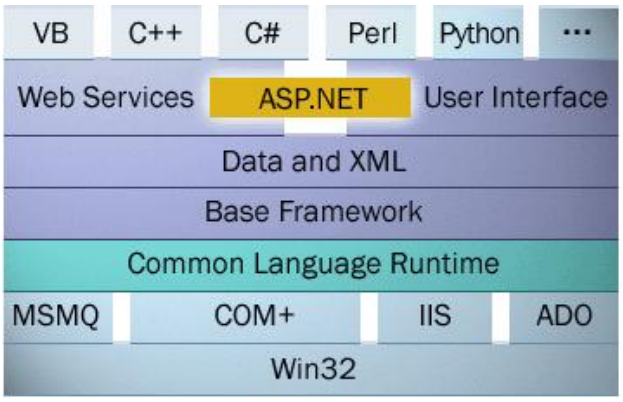


Figura 1.1: Lo stack delle tecnologie

1.2 CLR (Common Language Runtime)

CLR è responsabile dei servizi di run-time, tra cui l'integrazione del linguaggio, l'applicazione della protezione e la gestione della memoria, dei processi e dei thread. Svolge, inoltre, un importante ruolo in fase di sviluppo in quanto grazie a particolari funzionalità, tra cui la gestione della security, l'assegnazione di nomi sicuri, la gestione delle eccezioni tra più linguaggi, l'associazione dei dati dinamica e altre funzioni, consente agli sviluppatori di scrivere una minore quantità di codice per convertire la logica business in un componente riutilizzabile.

Il run-time è responsabile della gestione del codice e di fornire servizi al codice stesso durante la sua esecuzione. I linguaggi di programmazione .NET, tra cui Visual Basic .NET, Microsoft Visual `C#`^{TM}, le estensioni gestite C++ e molti altri linguaggi di fornitori diversi, utilizzano i servizi e le funzionalità .NET mediante un set di classi unificate.

La base del funzionamento del riutilizzo binario del codice in .NET è costituita dal Common Language Runtime, un ambiente d'esecuzione comune per tutti i linguaggi.

Il codice eseguito dal runtime parte dalla compilazione del codice intermedio detto IL (Intermediate Language). Tutti i compilatori dei linguaggi del CLR compilano in IL e quindi dispongono di un ambiente d'esecuzione comune con tipi di dati comuni, Figura 1.2.

In fase di esecuzione il linguaggio intermedio viene compilato in codice nativo della piattaforma sulla quale .NET è in esecuzione, Figura 1.3, grazie ad un JITer (Just in Time Compiler). La compilazione avviene in fase di runtime, solo per le parti di codice eseguite e, generalmente, una sola volta nel caso in

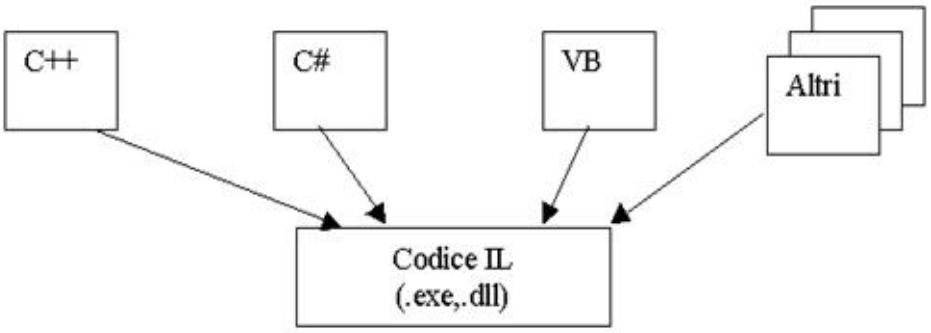


Figura 1.2: Tutti i linguaggi del CLR compilano in IL

cui la stessa parte di codice venga eseguita più volte nell'ambito della stessa istanza.

Questa tecnica di compilazione garantisce numerosi vantaggi, tra cui:

1. Il codine nativo viene compilato in base alla tipologia della macchina su cui viene eseguito: questo permette, ad esempio, un ottimizzazione del codice su macchine IA-64

2. Vengono compilate solo le parti del codice effettivamente utilizzate

1.3 La libreria di classi

Le classi unificate .NET rappresentano la base su cui vengono create le applicazioni indipendentemente dal linguaggio utilizzato. Le classi unificate vengono utilizzate sia per la concatenazione di una stringa, sia per creare un servizio Windows, o un'applicazione Web su più livelli.

Nel loro insieme, le classi offrono un'interfaccia di sviluppo comune e coerente per tutti i linguaggi supportati da .NET Framework.

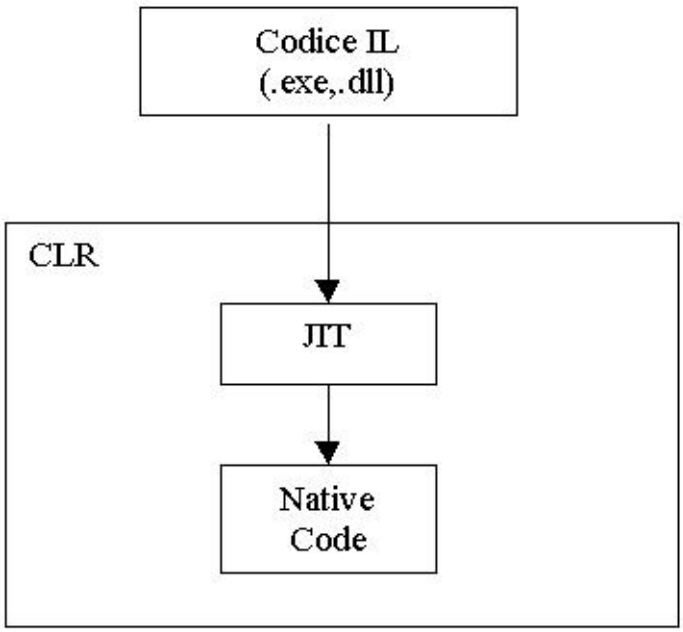


Figura 1.3: Solo in fase di esecuzione il codice IL viene compilato in codice nativo

1.4 Assembly, Tipi e Namespace

Le unità binarie in .NET prendono il nome di Assembly, indipendentemente dal fatto che si tratti di .exe o .dll.

Alla base del contenuto degli assembly abbiamo i Tipi che possiamo utilizzare nella programmazione .NET (Strutture, Classi, Delegates, etc).

Un assembly può essere considerato come un insieme di Tipi e risorse unitariamente organizzati e descritti con uno specifico entry point che può essere eseguito o utilizzato per accedere ai tipi esportati (exe o dll).

I tipi all'interno di un Assembly sono organizzati in Namespace, che li raggruppano e ne semplificano l'organizzazione e l'utilizzo. Gli stessi tipi contenuti nelle classi di base di .NET sono organizzati in Namespace.

I Namespace permettono anche un organizzazione gerarchica. Ad esempio il Namespace principale dei servizi base è contenuto in System, i tipi per la costruzione delle finestre Windows sono contenuti nel Namespace WinForms che a sua volta è contenuto in System.

Una delle caratteristiche più importanti degli Assembly sta nella loro capacità di autodescriversi. Questa caratteristica è alla base della semplificazione della gestione del versioning e della distribuzione delle applicazioni. In ogni unità binaria .NET, infatti, esistono metadati che consentono la completa descrizione della struttura del componente e dei tipi in esso contenuti e i riferimenti a risorse e assembly che gli occorrono per poter funzionare (ovvero gli assembly da cui è dipendente).

Il file di un assembly è essenzialmente strutturato come rappresentato nella Figura 1.4.

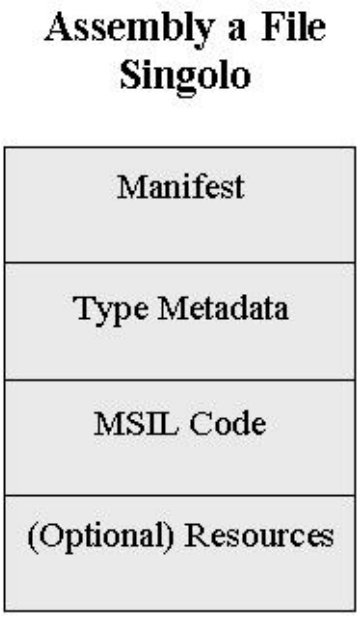


Figura 1.4: Formato di un Assembly .NET

Il manifesto contiene le informazioni di dipendenza da altri assembly e descrive caratteristiche e versione dei fornitori di servizi. Nel manifesto, poi, abbiamo le informazioni relative al componente vero e proprio, ovvero versione, originatore, etc.

La parte di Type Metadata contiene informazioni relative ai tipi contenuti nell'assembly. Questa parte può essere estesa con l'inserimento di attributi personalizzati.

Un assembly può essere costituito anche da più file binari, in questo caso l'architettura conterrà sempre un solo manifesto e sarà distribuito su più elementi binari di tipi e risorse.

Capitolo 2

All'interno del CLR

2.1 Microsoft .Net Framework: Il CLR

È difficile parlare di CLR senza discutere di differenza fra le specifiche e l'implementazione. Come parte dell'iniziativa .NET, Microsoft ha presentato gran parte della piattaforma a varie organizzazioni che si occupano di ratificare standard. In particolare, Microsoft ha presentato la Common Language Infrastructure (CLI) all'ECMA[1].

2.2 Common Language Infrastructure (CLI): Architettura

Fornisce le specifiche relative al codice eseguibile e all'ambiente di esecuzione. Il CLI può essere suddiviso in questo modo:

[1] http://www.ecma-international.org/

- CTS (Common Types System)
- MetaData
- CLS (Common Language Specification)
- VES (Virtual Execution System)

Il cuore del CLI è il CTS (descritto più avanti) il quale è condiviso dai compilatori, dagli strumenti e dallo stesso CLI. Tutti questi aspetti del CLI formano un framework comune per la progettazione, lo sviluppo e l'esecuzione di applicazioni.

L'obiettivo del CLI è di rendere più facile lo sviluppo di componenti ed applicazioni da qualsiasi linguaggio; raggiunge questo obiettivo definendo un insieme standard di tipi (qualcosa di simile al set di classi Java) e rendendo ogni componente completamente autodescrittivo. Le specifiche di tutto il CLI sono state rese pubbliche e standardizzate da ECMA standard 335 [3]

2.3 CTS (Common Types System)

Il CTS stabilisce un framework che permette l'integrazione cross-language, la sicurezza dei tipi e alte performance nell'esecuzione del codice. Il CTS è il modello che definisce le regole che il CLI segue quando dichiara, usa e gestisce i tipi.

Di fatto dà delle definizioni su cosa significa tipo, classe astratta e non, interfaccia, sul significato del cast, sulla visibilità, accessibilità, sicurezza e scope di metodi, sulle deleghe, etc.

2.4 MetaData

Contiene tutte le informazioni usate dal CLI per localizzare e caricare classi, le loro istanze in memoria, per risolvere l'invocazione di metodi, per trasformare MSIL in codice nativo garantendo la sicurezza e definendo il contesto.

Il MetaData è mantenuto (in forma persistente) in maniera indipendente dal linguaggio di programmazione usato.

2.5 CLS (Common Language Specification)

Il CLS è un sottoinsieme delle regole CTS volte all'interoperabilità dei linguaggi. Consiste in un insieme di regole e vincoli (utili per chi scrive i compilatori di linguaggi che vogliono essere conformi alle specifiche CLS e per chi scrive librerie) che permettono alle librerie di essere completamente fruibili da qualsiasi linguaggio che supporta il CLS.

Ogni tipo generato per essere eseguito su una implementazione del CLI deve essere conforme alle specifiche del CLS. I programmatori così possono scrivere codice che può essere usato in maniera del tutto trasparente da altri programmatori che usano altri linguaggi. Gli sviluppatori che progettano API che sottostanno agli standard CLS, non si dovranno preoccupare del linguaggio per cui queste API sono state sviluppate poiché saranno accessibili da tutti i linguaggi.

2.6 VES (Virtual Execution System)

Fornisce un ambiente di esecuzione per "managed code". Si può definire come la "Virtual Machine" di Visual Studio .Net e come tale carica, esegue e gestisce il codice MSIL; oltre al normale supporto che una virtual machine offre, fornisce anche un modello di eccezioni.

2.7 Il CIL

Mentre il CTS definisce un completo sistema dei tipi ed il CLS ne specifica un subset, il Common Intermediate Language specifica un set di tipi ancora più ristretto. Questi tipi sono chiamati "Tipi base del CLI" e sono:

1. Un sottoinsieme dei tipi numerici (int32, int64, native int, F)
2. Riferimenti ad oggetti, senza distinzione tra i tipi di oggetti riferiti
3. Tipi puntatore senza distinzione tra i tipi puntati

I riferimenti agli oggetti sono completamente opachi. Non hanno istruzioni aritmetiche che permettono di avere come operandi riferimenti ad oggetti. Le uniche operazione permesse sono quelle di identità ed equivalenza.

Ci sono due tipi di puntatori:

1. I puntatori gestiti: possono puntare ad una variabile, un campo di un oggetto etc. I puntatori gestiti non possono essere Null.
2. I puntatori non gestiti: sono i tradizionali puntatori usati in C o C++. Non hanno restrizioni sul loro uso. I puntatori non gestiti non sono gestiti dal Garbage Collector.

2.8 Verifica del codice

La memory safety è una proprietà che assicura che i programmi che vengono eseguiti nello stesso spazio di memoria siano correttamente isolati dagli altri.

Il CIL è in grado di produrre codice verificabile e non. Molte istruzioni del CIL non sono verificabili, per queste istruzioni non è garantito il corretto utilizzo della memoria.

L'algoritmo di verifica fornisce dettagliate condizioni di utilizzo delle istruzioni CIL.

2.9 Algoritmo di verifica

Le varie implementazioni del CLI di Microsoft usano differenti versioni dell'algoritmo di verifica. Tuttavia l'utility PEVerify.exe implementa una versione portabile dell'algoritmo come specificato nello standard ECMA.

L'algoritmo tenta di associare uno stato valido dello stack per ogni istruzione CIL. Lo stato dello stack specifica il numero di slot ed il proprio tipo.

L'algoritmo assume che il CLI azzeri tutta la memoria tranne che l'evaluation stack prima che questa sia visibile al programma. Successivamente l'algoritmo simula tutte le possibili esecuzioni e verifica che siano rispettate le specifiche dello stack. Ad ogni passo l'algoritmo verifica le istruzioni CIL per stabilire il nuovo stato dello stack.

Capitolo 3

Microsoft .NET: sicurezza

3.1 Le basi per la sicurezza

Le basi per la sicurezza dell'architettura .NET sono radicate nel CLR e prendono il nome di Esecuzione Gestita.

Durante l'esecuzione gestita il CLR ha la completa conoscenza di tutti gli aspetti del programma in esecuzione. Questo significa conoscere lo stato e la vita delle variabili locali in un metodo. Significa anche conoscere la provenienza del codice per ognuno degli stack frame. Tutto ciò permette di conoscere tutti gli oggetti esistenti ed i riferimenti agli oggetti, incluse le informazioni sulla raggiungibilità.

Il codice gestito viene accuratamente verificato dal Loader degli Assembly per garantirne la correttezza a livello di gestione dei tipi, nonché per accertare il corretto funzionamento di altre proprietà rispetto agli standard definiti.

Questo tipo di protezione rende teoricamente impossibili tentativi di violazione della security con attacchi di tipo buffer overflows o riferimenti non

autorizzati ad aree di memoria private.

Se questo è il meccanismo di base per garantire la cosiddetta Type Safety in applicazioni .NET, bisogna sapere che la politica di protezione è ben più articolata, si avvale di meccanismi di crittografia e firma digitale ed utilizza, inoltre, i seguenti meccanismi di protezione:

1. Evidence Based Security
2. Code Access Security
3. Role Based Security
4. Isolate Storage

3.2 Evidence Based Security

Gli elementi chiave del modello di sicurezza basato sulle prove sono costituiti dalle Evidence (prove dell'assembly), le Policy (politiche di sicurezza) e le Permissions (Autorizzazioni).

3.2.1 Evidence

Il termine prove indica semplicemente le informazioni relative al codice passate ai criteri di protezione. Si tratta fondamentalmente delle risposte al gruppo di domande generalmente poste tramite i criteri di protezione:

- Da quale sito proviene l'assembly?
- Da quale URL proviene l'assembly? I criteri di protezione richiedono l'indirizzo specifico da cui è stato scaricato l'assembly.

- Da quale area proviene l'assembly? Le aree sono descrizioni dei criteri di protezione, ad esempio Internet, Intranet, computer locale e così via, basate sulla posizione del codice.

- Qual è il nome sicuro dell'assembly? Il nome sicuro (Strong Name) è un identificatore sicuro dal punto di vista crittografico, fornito dall'autore dell'assembly. Sebbene non rappresenti in effetti un meccanismo di autenticazione dell'autore, identifica in modo univoco l'assembly e garantisce che non sia stato contraffatto.

3.2.2 Policy

Le Policy, fondamentalmente stabiliscono quali risorse possono essere accedute da un assembly, prevenendo che erroneamente, o con intenzione malevole venga corrotta l'integrità del sistema.

La funzione basilare dei criteri di protezione è quella di stabilire una corrispondenza tra le prove dell'assembly ed il sistema di autorizzazioni.

3.2.3 Permissions

Le autorizzazioni sono alla base del sistema di protezione. Stabiliscono una corrispondenza tra le risorse ed i diritti di accesso sulla base delle prove ricavata dall'assembly. Il Framework .NET stabilisce permessi per vari oggetti tra cui: FileIO, Environment, FileDialog, Registry, Socket, Web, Isolate Storage etc.

È possibile che l'autore di un assembly sappia che per il corretto funzionamento è necessario un set minimo di autorizzazioni oppure che l'assembly non richiederà mai determinate autorizzazioni. Queste informazioni aggiuntive sui

requisiti possono essere passate al sistema di criteri tramite una o più richieste di autorizzazioni specifiche.

3.3 Code Access Security

Il sistema di protezione di accesso al codice garantisce che il codice di un assembly non tenti di violare i permessi di esecuzione assegnati.

I permessi di esecuzione vengono stabiliti al caricamento di assembly in memoria. Se un metodo dell'assembly tenta di accedere ad una risorsa, viene avviata una procedura di verifica dei diritti di accesso, ispezionando tutta la catena delle chiamate.

La Figura 3.1 illustra le dinamiche di questo processo. L'assembly "A3" fornisce le proprie prove all'analizzatore dei criteri. L'analizzatore prende, inoltre, in considerazione le richieste di autorizzazioni dell'assembly per creare il gruppo di autorizzazioni concesse "G3". L'assembly A3 viene richiamato dall'assembly A2, che è stato richiamato dall'assembly A1. Quando l'assembly A3 esegue un'operazione che attiva un controllo di sicurezza, vengono esaminate anche le autorizzazioni concesse agli assembly A2 e A1 per assicurarsi che dispongano delle autorizzazioni richieste da A3. Nell'ambito di questo processo, definito **ispezione dello stack**, vengono esaminate le autorizzazioni concesse a ogni assembly nello stack per verificare che il set contenga l'autorizzazione richiesta dal controllo di sicurezza. Se ad ogni assembly dello stack è stata assegnata l'autorizzazione richiesta dal controllo di sicurezza, la chiamata ha esito positivo. Se viene rilevata l'assenza di un'autorizzazione richiesta per qualsiasi

assembly, l'ispezione dello stack viene interrotta e verrà generata un'eccezione di protezione.

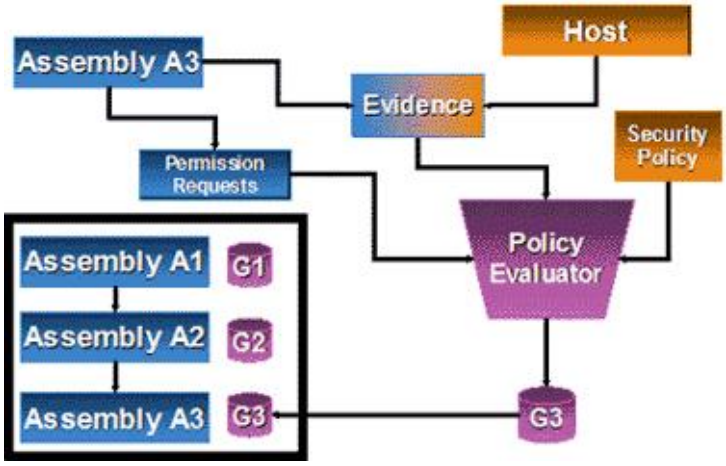


Figura 3.1: Processo di ispezione dello stack, per la valutazione dei diritti di accesso ad una risorsa.

3.4 Role Based Security

A volte è opportuno che le decisioni relative alle autorizzazioni vengano basate sull'identità autenticata o sul ruolo associato al contesto di esecuzione del codice. I servizi disponibili in .NET Framework consentono di incorporare questo tipo di logica nelle applicazioni in modo semplice, sulla base dei concetti di identità e principal.

Le identità possono corrispondere all'utente connesso al sistema operativo o essere definite dall'applicazione. Il principal corrispondente include l'identità oltre a tutte le informazioni eventualmente correlate al ruolo, ad esempio il gruppo dell'utente definito dal sistema operativo.

3.5 Spazi di archiviazione isolati: Isolate Storage

In .NET Framework è disponibile una speciale infrastruttura denominata archiviazione isolata, un meccanismo molto simile alla SandBox di Java, che supporta l'archiviazione dei dati anche nel caso non sia consentito l'accesso ai file. Ad esempio, se si scarica e si esegue un **controllo gestito** da Internet, a tale controllo viene concesso un set di autorizzazioni limitato, ma non il diritto di lettura o scrittura dei file.

L'infrastruttura di archiviazione isolata viene implementata tramite un nuovo gruppo di tipi e metodi supportati da .NET Framework per l'archiviazione locale. Sostanzialmente, ad ogni assembly viene concesso l'accesso a uno spazio di archiviazione isolato su disco. Non viene permesso l'accesso ad altri dati e ogni spazio di archiviazione isolato è disponibile esclusivamente per l'assembly specifico per cui è stato creato.

Le applicazioni possono avvalersi di questi spazi di archiviazione isolati per la memorizzazione dei log, il salvataggio delle impostazioni, oppure per il salvataggio su disco dei dati sullo stato per utilizzi successivi. Poiché la posizione dello spazio di archiviazione isolato è predeterminata, questo strumento risulta piuttosto utile per definire spazi di archiviazione univoci senza dover definire percorsi specifici per i file.

Anche per il codice di Intranet locali sono previste restrizioni simili, anche se meno limitative, e per tale codice è disponibile una quota maggiore dello spazio di archiviazione isolato. Infine, il codice proveniente dall'area Siti con

restrizioni (ovvero siti considerati non affidabili) non ottiene l'accesso allo spazio di archiviazione isolato.

Parte II

Riservatezza delle informazioni in Microsoft .NET

Capitolo 4

I limiti della sicurezza in .NET

4.1 Introduzione

Abbiamo analizzato in dettaglio le politiche di sicurezza adottate da .NET Framework per garantire un'esecuzione sicura delle applicazioni. In particolare dovrebbe essere ben chiaro come alla base di tutto ci debba essere l'esecuzione gestita.

Codice giudicato non sicuro, durante la fase di caricamento, deve essere esplicitamente autorizzato dall'amministratore di sistema, che dovrà provvedere a dare trust al programma, se desidera che questo esplichi le sue funzioni.

Un esecuzione gestita garantisce l'utilizzo di codice verificabile, questo significa che il comportamento di ogni istruzione non viola l'integrità del sistema e che non ci saranno accessi indesiderati ad aree di memoria proibite.

Se questo non dovesse bastare, il CLR è di grado di analizzare la provenienza e l'integrità del codice .NET: se questo dovesse risultare particolarmente pericoloso è in grado di relegare la sua esecuzione all'interno di uno spazio di

esecuzione isolata.

4.2 Il problema

Le politiche di sicurezza adottate in .NET permettono un meccanismo di controllo degli accessi efficace ed articolato. Tuttavia nel contesto di un applicazione autorizzata all'esecuzione, che esegue solo operazioni consentite, qual'è l'effettivo controllo sulle informazioni manipolate?

Alcune applicazioni hanno la necessità di accedere a files privati, o di manipolare informazioni personali per fornire all'utente servizi di utilità, quest'ultimo vorrebbe permettere la lettura dei propri dati, ma impedire la pubblicazione degli stessi.

In questo contesto una politica di sicurezza basata sul controllo degli accessi si rivela inefficace e richiede dei meccanismi di protezione più affidabili, come un analisi statica del codice atta al controllo dei flussi di informazioni.

Per chiarire meglio la problematica consideriamo una semplice utility per il calcolo del Codice Fiscale [1].

Si tratta di una applicazione molto utile, scaricabile gratuitamente da vari siti, che richiede l'inserimento di molti dati personali, tra cui la data di nascita, il comune di nascita etc.

Il programma in Figura 4.1, realizzato in `C#`, ad uso esclusivamente dimostra-

[1] Il programma d'esempio è scaricabile gratuitamente presso:
http://www.nicoprovenzano.it/secureflow/codfisc.exe

Figura 4.1: Programma per il calcolo del codice fiscale, implementato come controllo gestito

tivo, è stato implementato e reso disponibile in due forme differenti: **Controllo gestito**[2] e **Windows Form**[3].

4.3 Il programma WinForm .NET

È, ormai, pratica comune scaricare dalla rete software di qualunque tipo. Il Web è molto ricco di software gratuiti in grado di sopperire alle più svariate esigenze. Il programma realizzato rispecchia proprio questa esigenza promettendo all'incauto utente di poter comodamente calcolare il codice fiscale dei propri clienti. Una volta scaricato sul proprio computer ed eseguito, il programma ha, potenzialmente, il controllo completo del computer: il Framework .NET

[2]Il controllo gestito è l'equivalente delle applet Java, viene eseguito direttamente all'interno della finestra del Browser

[3]Una Windows Form è una applicazione .NET a finestre.

permette, infatti, l'esecuzioni di codice di qualunque tipo, purché type safe, ai programmi eseguiti in modalità Off-line.

È ovvio che una situazione di questo tipo non necessita di ulteriori chiarimenti: il programma scaricato si comporta come un vero e proprio Trojan Horse che, una volta eseguito, potrebbe fare qualsiasi cosa.

Il programma realizzato, include funzionalità simili a quelle descritte in § 4.4, inoltre fa vedere come sia possibile, a totale insaputa dell'utente, inviare in rete il contenuto di alcuni file di sistema.

4.4 Il controllo gestito

È stato particolarmente complicato reperire informazioni sui controlli gestiti .NET e la loro modalità di esecuzione. Come spiegato in [11] un controllo gestito .NET è semplicemente un Windows Form .NET. Per abilitare la sua esecuzione in modalità, per così dire Applet, sono necessarie le seguenti operazioni:

1. Allestire un server IIS 6.0[4]
2. Creare una Virtual Directory[5], con diritti di esecuzione, in cui depositare l'applicazione .NET
3. Installare sul server il Framework .NET
4. Creare una pagina Html per il contenimento del controllo gestito

[4]Internet Information Services 6.0 è il servizio di web publishing integrato in Windows Server 2003

[5]Con questo nome si designa una cartella del file system destinata a ospitare siti o applicazioni web

5. Aggiungere alla pagina web il controllo gestito mediante la seguente sequenza di istruzioni:

```
<object id="myComponent"
classid="http:[assembly URL]#[full class name]"
height="value" width="value">
</object>
```

Con una logica di funzionamento molto simile a quella delle Applet Java, un controllo gestito viene scaricato sull'host ed eseguito dal CLR in modalità isolata con più o meno diritti di accesso alle risorse, a seconda che il sito provenga dalla Intranet o da Internet. La grande utilità di una gestione di questo tipo risiede nel fatto che il CLR è in grado di avviare l'applicazione in modalità Windowed, ossia completamente integrata con la pagina Web di riferimento. Confrontando, infatti, la Figura 4.1 con la Figura 4.2, ci si rende conto che non c'è alcun modo per capire che la pagina contiene un programma. Al contrario di Java, che nei casi più semplici avvisa l'utente della presenza di un'Applet visualizzando nel Tray Icon un'apposita icona, il Framework .NET non dà alcun segno del caricamento di un controllo gestito, questo per il semplice motivo che è assolutamente sicuro di aver relegato il controllo in un area isolata.

Agendo in questa maniera è possibile, in modo semplice e senza impiegare logica Server-Side, implementare interfacce utente eleganti e ricche di funzionalità, con un impegno aggiuntivo minimo rispetto alla parallela progettazione di un interfaccia ad esecuzione locale.

Figura 4.2: Il controllo gestito si trova all'interno dell'area rossa punteggiata

4.4.1 Funzionamento del programma d'esempio

La logica di funzionamento di un programma di calcolo del Codice Fiscale è molto semplice ed è per questo motivo che è stato scelto come esempio per la problematica della tutela della riservatezza delle informazioni.

L'infrastruttura allestita per il funzionamento del programma è mostrata in Figura 4.3 e si avvale dei seguenti componenti:

- una pagina Web, in modalità Client-Side, che ospita il controllo gestito ed è adibita all'interfacciamento con l'utente
- un database remoto, utilizzato per la gestione dei dati utente
- un insieme di applicazioni Server-Side, necessarie per il funzionamento del programma ed altre funzionalità non strettamente necessarie.

I dati inseriti dall'utente vengono inviati al server remoto per il calcolo del

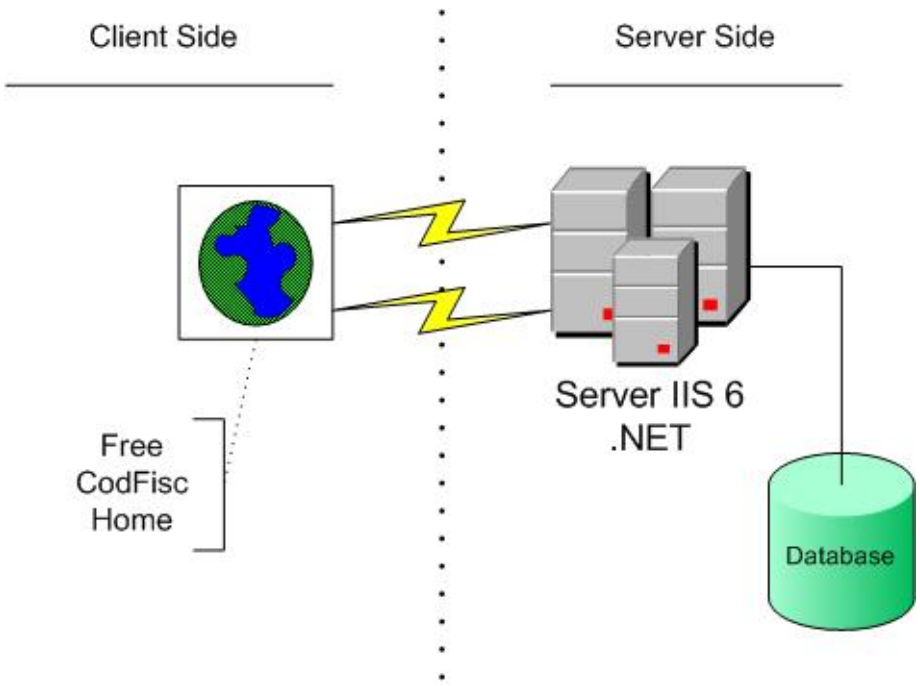


Figura 4.3: Infrastruttura necessaria al funzionamento del programma

Codice Fiscale. Questa tecnica è utilizzata per carpire i dati privati dell'utente, Figura 4.4. Il Controllo Gestito è stato appositamente progettato per permettere un testing efficace dell'applicazione, come spiegato in § 6. Per questo motivo all'interno del programma viene creata una struttura dati adibita al contenimento dei dati utente.

Il contenuto di questa struttura dati viene, successivamente, inviato ad una pagina realizzata in tecnologia ASPX[6], in grado di eseguire sul server remoto, di fare tutte le elaborazioni necessarie, tra cui memorizzare i dati dell'utente, e di spedire indietro il codice fiscale calcolato.

Il programma, inoltre, tenta di inviare, soltanto, il nome del comune per verificare, nel database remoto, che sia un nome valido.

[6]Active Server Pages .NET

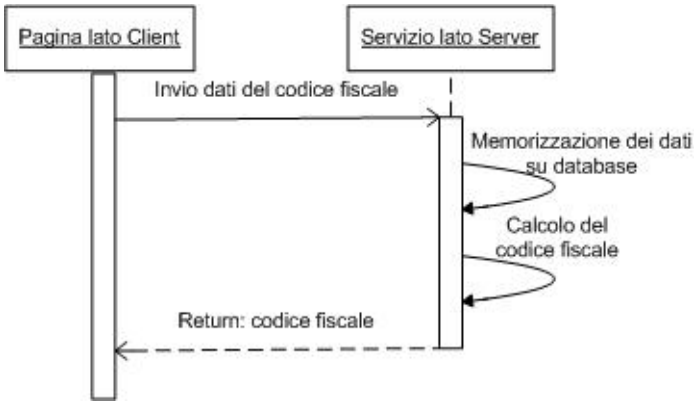


Figura 4.4: I dati, inviati dal client, vengono intercettati dal server remoto, che, per non destare sospetto, ritorna effettivamente il codice fiscale

È evidente come l'uso improprio di questa architettura software permetta, in modo semplice e nel totale rispetto dei meccanismi di protezione, una violazione della privacy, consentendo l'invio indiscriminato di informazioni personali. Un sistema di protezione basato esclusivamente sulle tecnologie precedentemente riassunte, in primis lo storage isolato, non permette di risolvere una situazione di questo tipo. Una possibile soluzione sarebbe quella di restringere, ulteriormente, la politica di sicurezza, impedendo a programmi scaricati dalla rete di accedere ai socket, tuttavia un approccio di questo tipo risulterebbe molto limitativo e non darebbe più senso ad applicazioni di questo tipo.

Si rende quindi necessario l'utilizzo di nuovi meccanismi come l'analisi statica del codice, atti a controllare il flusso delle informazioni. Possiamo, ad esempio, assegnare un alto livello di segretezza alle informazioni ritenute private, ad esempio la struttura dati che contiene i dati dell'utente, ed un basso livello di segretezza alle informazioni inviate sui socket: in questo modo un tentativo di invio di dati personali risulterebbe in un assegnamento di dati ad alto livello di sicurezza verso dati a basso livello di segretezza e quindi ad alto rischio. Un

problema di questo tipo sarebbe chiaramente sintomo di una violazione della Privacy.

Per rilevare flussi illeciti di informazioni in Assembly .NET in questa tesi viene proposto uno strumento chiamato SIFDotNet basato sull'analisi statica del codice.

Come è possibile vedere nel Capitolo 7 dedicato ai test, impostando correttamente i livelli di sicurezza, SIFDotNet è in grado di rilevare la presenza di connessione remote, da parte del controllo gestito, e se questo non fosse sufficiente, è anche in grado di rilevare correttamente una violazione della Privacy in seguito al tentativo di invio della struttura dati personale, considera, invece, correttamente, l'invio di informazioni meno importanti.

Un approccio di questo tipo potrebbe, ad esempio, allertare l'incauto utente, che si interrogherebbe sulla necessità di calcolare il Codice Fiscale in questa maniera complessa, piuttosto che effettuare il calcolo sul computer di sua proprietà.

Capitolo 5

Flusso sicuro delle informazioni

5.1 Introduzione

Nei precedenti capitoli è stato affrontato il problema della sicurezza in applicazioni .NET. Si è visto che il modello di sicurezza di Microsoft .NET è un valido strumento di protezione dell'integrità del sistema. Questo modello di sicurezza, tuttavia, non è in grado di garantire controlli sulla propagazione di informazioni private a cui è stato concesso l'accesso. In questo capitolo si espone una breve descrizione sull'approccio per la verifica del flusso di informazione sicuro, sviluppato in [1] e rielaborato per tener conto dell'architettura .NET.

Consideriamo il sottoinsieme del linguaggio MSIL illustrato in Figura 5.1, dove le istruzioni sono raggruppate in base alla funzione espletata (il set di istruzioni MSIL effettivamente utilizzato è illustrato nel § 5.2).

Il CLR gestisce le informazioni facendo uso di uno stack degli operandi, detto *Evaluation Stack*, manipola un insieme di registri locali ed un insieme di registri per gli argomenti di ogni metodo, ed uno heap contenente le istanze

pop	Rimuove il valore al top dello stack.
dup	Copia il valore corrente al top dello stack ed inserisce la copia nello stack.
op	Rimuove due operandi dallo stack, effettua, quindi, l'operazione ***op*** $\in$ { add, mull, div .. }
const d	Inserisce la costante d nello stack, dove ***const*** $\in$ { ldc.i4, ldc.r4, .. }
load x	Inserisce il valore del registro x in cima allo stack, dove ***load*** $\in$ { ldc.i4, ldc.r4, .. }
store x	Preleva un valore dalla top dello stack e lo memorizza nel registro locale x, dove ***store*** $\in$ { stloc, starg, .. }
if*cond* j	Preleva un valore dal top dello stack e lo confronta con la condizione *cond*, dove ***if****cond* $\in$ { beq, bge, .. }. Salta a j se il risultato è falso.
goto j	Salta a j, dove ***goto*** $\in$ { br, .. }
ldfld $C.f$	Preleva un riferimento ad un oggetto di classe C dal top dello stack; preleva il campo f da tale oggetto e lo inserisce in cima allo stack
stfld $C.f$	Preleva dal top dello stack un valore v ed un riferimento ad un oggetto di classe C; inserisce il valore v nel campo f dell'oggetto
invoke $C.mt$	Preleva i valori v_1,..,v_n ed un riferimento ad un oggetto di classe C dallo stack. Invoca il metodo $C.mt$ dell'oggetto riferito con i parametri attuali v_1,..,v_n, dove ***invoke*** $\in$ { call, callvirt }
return	Restituisce come valore di ritorno dalla chiamata del metodo il top dello stack.

Figura 5.1: Classificazione delle istruzioni IL

degli oggetti.

Le istruzioni sono *politipiche*, questo vuol dire che esiste un'unica operazione (**ADD** per effettuare la somma di due interi, due reali etc..).

Durante la nostra trattazione prenderemo in considerazione Assembly .NET realizzati a partire dal set di istruzioni **SifIL**, mostrato in § 5.2, considereremo metodi dotati di un qualunque numero di argomenti e forniti di un valore di ritorno. Prenderemo in considerazione, inoltre, Assembly .NET dotati di più Tipi, anche annidati. Non considereremo la gestione delle eccezioni. Questa funzionalità sarà aggiunta in futuro.

Il codice CIL di un metodo è una sequenza β di istruzioni. Quando viene invocato un metodo, (istruzioni **call, callvirt**) ad esso è associato un nuovo stack vuoto ed una memoria iniziale in cui tutti i registri hanno valore indefinito eccetto il primo, registro 0, che contiene un riferimento all'istanza dell'oggetto a cui appartiene il metodo chiamato, ed i successivi n registri, numerati da 1 ad $n + 1$, che contengono i parametri attuali. Quando un metodo termina, il controllo è restituito al chiamante: l'ambiente di esecuzione del chiamante (stack degli operandi e registri locali) viene ripristinato ed il valore di ritorno, se esiste, viene inserito in testa allo stack.

5.2 Il set di istruzioni SifIL

Il Common Intermediate Language (CIL) è, come spiegato in § 2.7, il set di istruzioni object-oriented con cui vengono distribuiti gli Assembly .NET. CIL supporta numerosi costrutti di alto livello, inclusa l'ereditarietà, la garbage collection ed i meccanismi di security e type safety. Sono inclusi poi tutta una

serie di costrutti come i puntatori non gestiti ed altre istruzioni che non sono verificabili, ma sono presenti solo per garantire il supporto di compilatori che, palesemente, non sono Type-Safe come quello *C++*.

Rivolgeremo la nostra attenzione ad un frammento, abbastanza grande, del CIL, che include **solo istruzioni verificabili** e, quindi, necessarie per la costruzione di programmi sicuri dal punto di vista della gestione dei tipi e della memoria.

Nel frammento che abbiamo nominato SifIL, sono presenti alcune istruzioni dotate di un suffisso *.un*, il quale indica la gestione di un valore unsigned. Altre istruzioni sono dotate del suffisso *.ovf*, che indica la generazione di un'eccezione in presenza di overflow. Altre istruzioni, inoltre, sono dotate di suffissi tipo *.i1, .i2, .i3, .i4, .i8, .u1, .u2, .u4, .u8, .r4, .r8, .r.un, .i, .u, .ref*, che stanno ad indicare il tipo del risultato. Tutte le istruzioni sono **verificabili**, ossia il verificatore CLR giudica Type Safe e Managed Assemblies programmi realizzati mediante queste istruzioni. Le istruzioni supportate sono mostrate nelle Figure 5.2 - 5.6. Assumiamo che *Value1, Value2* siano gli, eventuali, operandi delle istruzioni.

`ceq Push`	1 se value1 è uguale a value2, altrimenti 0
`cgt Push`	1 se value1 è maggiore di value2, altrimenti 0
`cgt.un`	Push 1 se value1 è maggiore di value2, quando vengono confrontati interi unsigned o unordered floating point, altrimenti 0
`clt Push`	1 se value1 è minore di value2, altrimenti 0
`clt.un`	Push 1 se value1 è minore di value2, quando vengono confrontati interi unsigned o unordered floating point, altrimenti 0

Figura 5.2: SifIL: comparisons

`beq`	Effettua il salto se value1 è uguale a value2.
`bge`	Effettua il salto se value1 è maggiore o uguale a value2.
`bge.un`	Effettua il salto se value1 è maggiore o uguale a value2, quando vengono confrontati interi unsigned o unordered floating point.
`bgt`	Effettua il salto se value1 è maggiore di value2.
`bgt.un`	Effettua il salto se value1 è maggiore di value2, quando vengono confrontati interi unsigned o unordered floating point.
`ble`	Effettua il salto se value1 è minore o uguale a value2.
`ble.un`	Effettua il salto se value1 è minore o uguale a value2, quando vengono confrontati interi unsigned o unordered floating point.
`blt`	Effettua il salto se value1 è minore di value2.
`blt.un`	Effettua il salto se value1 è minore di value2, quando vengono confrontati interi unsigned o unordered floating point.
`bne.un`	Effettua il salto se value1 diverso da value2, quando vengono confrontati interi unsigned o unordered floating point.
`br`	Salto incondizionato.
`call`	Chiama il metodo indicato dal descrittore di metodo passato.
`callvirt`	Chiama un metodo ad associazione tardiva su un oggetto, inserendo il valore restituito dal metodo nello stack di valutazione.
`jmp`	Esce dal metodo corrente e passa a quello specificato.
`brtrue, brfalse`	Provoca il salto se al top dello stack è presente un valore intero pari a 0, o se è un puntatore impostato a null o un riferimento ad oggetto. brtrue fa il lavoro opposto.
`ret`	Ritorna dal metodo corrente. Il valore restituito, se presente, deve trovarsi al top dello stack.

Figura 5.3: SifIL: Branches

`add`	Somma value1 con value2 e mette il risultato nello stack. Non è considerato l'overflow per numeri interi. Per floating-point l'overflow ritorna +inf o -inf
`add.ovf`	Somma due interi con segno value1 con value2 e mette il risultato nello stack. Genera, in caso di overflow, una eccezione di tipo OverflowException
`add.ovf.un`	Somma due interi senza segno value1 con value2 e mette il risultato nello stack. Genera, in caso di overflow, una eccezione di tipo OverflowException
`div`	Divide value1 con value2 e ritorna il quoziente o un risultato di tipo floating-point
`div.un`	Divide due valori di tipo unsigned
`mul`	Moltiplica due valori
`mul.ovf`	Moltiplica due valori interi con segno e genera OverflowException se il risultato non rientra nella stessa dimensione
`mul.ovf.un`	Moltiplica due valori interi senza segno e genera OverflowException se il risultato non rientra nella stessa dimensione
`neg`	Nega un valore
`rem`	Divide due valori e inserisce il resto nello stack di valutazione.
`rem.un`	Divide due valori di tipo unsigned e inserisce il resto nello stack di valutazione.
`sub`	Sottrazione
`sub.ovf`	Sottrazione con overflow tra due interi con segno
`sub.ovf.un`	Sottrazione con overflow tra due interi senza segno

Figura 5.4: SifIL: operatori aritmetici

`ldarg`	Mette sullo stack l'n-esimo argomento (n è codificato nell'istruzione).
`ldarga`	Mette l'indirizzo dell'n-esimo argomento sullo stack.
`ldc`	Mette una costante numerica sullo stack.
`ldloc`	Mette sullo stack l'n-esima variabile locale.
`ldnull`	Mette sullo stack un valore null.
`starg`	Toglie dallo stack un valore e lo carica come n-esimo argomento.
`stloc`	Toglie dallo stack un valore e lo carica come n-esima variabile locale.
`ldfld`	Trova il valore di un campo nell'oggetto il cui riferimento si trova attualmente nello stack di valutazione.
`stfld`	Sostituisce il valore memorizzato nel campo di un riferimento a un oggetto o puntatore con un nuovo valore.
`ldstr`	Inserisce un nuovo riferimento a un oggetto rappresentazione formale di stringa nei metadati.

Figura 5.5: SifIL: Load e Store

`ldc.i4`	Inserisce un valore fornito di tipo int32 nello stack di valutazione come int32.
`ldc.i4.0`	Inserisce il valore integer 0 nello stack di valutazione come int32.
`ldc.i4.1..8`	Inserisce il valore integer 1 nello stack di valutazione come int32.
`ldc.i4.m1`	Inserisce il valore integer -1 nello stack di valutazione come int32.
`ldc.i4.s`	Inserisce il valore fornito int8 nello stack di valutazione come int32 (forma breve).
`ldc.i8`	Inserisce un valore fornito di tipo int64 nello stack di valutazione come int64.
`ldc.r4`	Inserisce un valore fornito di tipo float32 nello stack di valutazione come tipo F (float).
`ldc.r8`	Inserisce un valore fornito di tipo float64 nello stack di valutazione come tipo F (float).

Figura 5.6: SifIL: Istruzioni per caricare una costante nello stack

`pop`	Rimuove il valore attualmente all'inizio dello stack.
`dup`	Copia il valore corrente più in alto nello stack di valutazione e inserisce la copia nello stack di valutazione.
`nop`	Riempie lo spazio se i codici operativi sono corretti. Non viene eseguita alcuna operazione significativa sebbene possa essere eseguito un ciclo di elaborazione.
`newobj`	Crea un nuovo oggetto o una nuova istanza di un tipo di valore, inserendo un riferimento a un oggetto (di tipo O) nello stack di valutazione.

Figura 5.7: SifIL: Altre istruzioni

5.3 Il modello per la Sicurezza

Sia $(\Sigma, \sqsubseteq)$ un lattice di livelli di sicurezza con $\sqsubseteq$ la relazione di ordinamento parziale sui livelli: se $\sigma_1, \sigma_2 \in \Sigma$ e $\sigma_1 \sqsubseteq \sigma_2$ significa che σ_1 è un livello di sicurezza inferiore rispetto a σ_2. Per esempio, se i livelli sono H, per privato, ed L per pubblico abbiamo $L \sqsubseteq H$. Indichiamo con $\sqcup$ l'operatore di least upper bound fra due livelli: $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_1 \sqcup \sigma_2 = \sigma_2$. Assumo che σ_0 sia il security level più basso: $\sigma_0 \sqsubseteq \sigma$, $\forall \sigma \in \Sigma$.

Dato un programma ed un assegnamento di livelli di sicurezza alle informazioni a cui accede il programma, il programma soddisfa il Secure Information Flow se, alla terminazione del programma, informazioni ad un dato livello di sicurezza non si sono propagate in oggetti di livello inferiore.

Lo studio del SIF di un programma viene effettuato analizzando il SIF di ogni singolo metodo, assumendo che, quando verifichiamo un metodo, gli altri metodi soddisfano tale proprietà.

Dato un insieme di classi C, denotiamo con *Mtds(C)* i metodi definiti per le classi $\in C$. Sia *S:* $C \rightarrow \Sigma$ una specifica di sicurezza che associa un livello di sicurezza ad ogni classe C. Assumiamo che tutte le istanze e tutti gli attributi

di una classe abbiano il livello di sicurezza della classe.

Sia *P: Mtds(C)* $\rightarrow \Sigma$ un assegnamento di livelli di sicurezza ai parametri di ogni metodo $\in$ *Mtds(C)* e *R: Mtds(C)* $\rightarrow \Sigma$ un assegnamento di livelli di sicurezza al valore di ritorno di ogni metodo. Definiremo ora il concetto di Secure Information Flow per un metodo $\in$ *Mtds(C)*. Dato S, P, R, un metodo soddisfa SIF l'esecuzione del metodo a partire da argomenti e classi con livello di sicurezza dato da S, P, non propaga in oggetti a livello σ informazioni $\not\sqsubseteq \sigma$ e ritorna un valore $\sqsubseteq R$.

L'information flow nel codice CIL di un metodo può essere esplicito o implicito. Si parla di information flow esplicito nel caso di un assegnamento. Un flusso di informazioni si dice, invece, implicito quando viene usato un valore all'interno di una istruzione condizionale.

Consideriamo il codice CIL illustrato in Figura 5.8, corrispondente ad un metodo *mt* di una classe A. Supponiamo che il registro `x1` (argomento del metodo A.*mt*) contenga il riferimento di un oggetto di una classe B. Inoltre, assumiamo che il livello di sicurezza della classe B sia più alto di quello della classe A. Bisogna notare che il registro `x0` contiene un riferimento alla classe A. Al termine dell'esecuzione del codice CIL, il valore finale del campo *f1* dell'oggetto A è 0 o 1 a seconda del valore del campo *f2* dell'oggetto B.

Il codice d'esempio illustra la tipica situazione in cui si presenta un caso di flusso implicito di informazione. C'è una violazione perché controllando il valore finale di A.*f1* viene rivelata l'informazione sul valore di B.*f2*, che ha un livello di sicurezza più alto.

```
0 : ldloc  x0
1 : ldarg  x1
2 : ldfld  B.f2
3 : bge    6
4 : ldc.i4 0
5 : jmp    7
6 : ldc.i4 1
7 : stfld  A.f1
8 : ldc.i4 1
9 : ret
```

Figura 5.8: Un caso di flusso implicito in un set di istruzioni CIL

5.4 Il metodo

Assumiamo che il codice CIL cui verrà applicato l'algoritmo di verifica del flusso sicuro di informazioni sia corretto dal punto di vista della verifica standard.

Il verificatore di information flow prende in input un set C di classi, un assegnamento *S:* $C \rightarrow \Sigma$ di livelli di sicurezza alle classi e un assegnamento *P,R:* *mtds(C)* $\rightarrow \Sigma$ di livelli di sicurezza agli argomenti ed al valore di ritorno di ogni metodo. Il verificatore analizza un metodo alla volta, partendo dall'assunzione che gli altri metodi soddisfino i requisiti del Secure Information Flow. Durante l'esecuzione astratta del codice CIL, ogni valore viene rappresentato con il livello di sicurezza derivato dal least upper bound dei livelli di sicurezza dei flussi di informazioni, impliciti ed espliciti, da cui dipende.

I flussi impliciti vengono tenuti in considerazione eseguendo le istruzioni in un *Ambiente di Sicurezza*, che rappresenta il least upper bound dei livelli di sicurezza dei flussi impliciti da cui l'istruzione dipende. Poiché il codice CIL non è strutturato, per individuare la regione di influenza di un flusso implicito viene

utilizzato il Control Flow Graph del programma ed il concetto di *postdominator* [15].

Dato un insieme di istruzioni CIL, il control flow graph del codice è un grafo orientato (V, L), dove V è un insieme di nodi, uno per ogni istruzione e $L \subset V \times V$ contiene (i, j) se e solo se l'istruzione all'indirizzo j può essere eseguita subito dopo l'istruzione all'indirizzo i. Per semplicità consideriamo che il grafo orientato abbia un solo nodo finale.

Se $i, j \in V$, j *postdomina* i, indicato con $j\ pd\ i$, se $j \neq i$ e j è in tutti i path da i al nodo finale. j *postdomina immediatamente* i, indicato con $j = ipd(i)$, se $j\ pd\ i$ e non c'è nessun nodo r fatto in modo tale che $j\ pd\ r\ pd\ i$.

I dati che vendono manipolati in un ambiente di sicurezza σ hanno un livello di sicurezza almeno σ. Quando viene eseguita un istruzione di controllo, supponiamo all'indirizzo i, l'ambiente viene, eventualmente, aggiornato in modo tale da considerare il livello di sicurezza del valore testato. A questo punto tutte le istruzioni che si trovano all'interno dell'istruzione condizionale vengono eseguite con il nuovo ambiente di sicurezza. L'istruzione, invece, che viene raggiunta da tutti i cammini che si dipartono dall'istruzione condizionale, calcolata come prima, non dipende dal nuovo ambiente. Nel seguito useremo il concetto di dipendenza di controllo.

Siano i, j due nodi nel control flow graph: si dice che j ha una dipendenza di controllo da i se e solo se c'è un path per cui j postdomina tutti i nodi inclusi tra i e j nel path da i a j e j non postdomina i. Intuitivamente, i è una istruzione di controllo che determina se il flusso di controllo passa da j: l'istruzione i può influenzare direttamente l'esecuzione di j e, per questo motivo, si dice che c'è

una dipendenza di controllo di j da i. I nodi che hanno lo stesso insieme di dipendenze di controllo si trovano nella stessa *regione di controllo*. D'ora in poi denoteremo l'insieme delle dipendenze di controllo di i con C_i.

5.5 L'interprete

Il verificatore di secure information flow per codice CIL effettua un'interpretazione astratta del codice CIL di un metodo. Ad ogni istruzione i viene assegnato uno stato Q_i, che rappresenta lo stato in cui è eseguita l'istruzione i. Q_i e una tupla (M, S, E), dove $M : Registers \rightarrow \Sigma$ è una corrispondenza di registri locali con livelli di sicurezza (la memoria), $S \in \Sigma^*$ è una corrispondenza di elementi dell' *evaluation stack* con livelli di sicurezza (lo stack) ed E è l'insieme delle coppie (c, l), dove $c \in C_i$ e $l \in \Sigma$. Intuitivamente, se l'istruzione all'indirizzo i ha una dipendenza di controllo da c e la coppia (c, l) rappresenta il livello di sicurezza del flusso implicito di c.

Sono definite le seguenti operazioni sull'ambiente E:

$level(E) = \sqcup_h l_h, \forall (c_h, l_h) \in E$

$E \downarrow_C = \{(c_h, l_h) | (c_h, l_h) \in E \wedge c_h \in C\}$

$dep(E) = \{c_h | (c_h, l_h) \in E\}$

$E \sqcup (c, l) = \begin{cases} E/\{(c,l')\} \cup (c, l \sqcup l') & if\ \exists (c,l') \in E \\ E \cup \{(c,l)\} & altrimenti \end{cases}$

$E \sqcup E' = E'' \mid dep(E'') = dep(E), \forall c \in dep(E) \cap dep(E')$ se $(c, l) \in E, (c, l') \in E'$ allora $(c, l \sqcup l') \in E''$.

Di seguito definiamo l'operazione di *lub* su Memoria, Stack e Ambiente. L'operazione $M \sqcup M'$ è eseguita per ogni elemento della Memoria. Analogamente, $S \sqcup S'$ è eseguito per ogni singola locazione Stack. $E \sqcup E'$ genera un

nuovo ambiente E'' fatto in modo tale che E'' sia una copia di E aggiornata con il least upper bound dei livelli di sicurezza di tutte le dipendenze a comune. L'operazione di least upper bound sugli Ambienti non è commutativa.

Dati due stati $Q_i = (M, St, E)$ e $Q'_i = (M', St', E')$ l'operazione di least upper bound è definita come segue:

$(M, St, E) \sqcup (M', St', E') = (M \sqcup M', St \sqcup St', E \sqcup E')$.

È definita una relazione di parziale ordinamento sui domini degli stati di ogni istruzione. Questa relazione è indotta dalla corrispondente relazione di ordinamento fra livelli di sicurezza. Date due Memorie M_1 e M_2, $M_1 \sqsubseteq M_2$ se e solo se per ogni registro x, vale che $M_1(x) \sqsubseteq M_2(x)$. Il dominio degli Stack ha un Bottom Element $\perp_{St}$, che è $\sqsubseteq$ di tutti gli Stack. Dati due Stack S_1 e S_2 diversi da $\perp_{St}$, $S_1 \sqsubseteq S_2$ se e solo se S_1 e S_2 hanno la stessa lunghezza e ogni elemento in S_1 è $\sqsubseteq$ dell'elemento di S_2 che si trova nella stessa posizione. Stack di dimensioni differenti non sono correlati. Dati due ambienti E_1 e E_2, fatti in modo tale che $dep(E_1) = dep(E_2)$, $E_1 \sqsubseteq E_2$ se e solo se per ogni istruzione di controllo $c \in dep(E_1)$ se $(c, l) \in E_1, (c, l') \in E_2$ e $l \sqsubseteq l'$. Come conseguenza, $level(E_1) \sqsubseteq level(E_2)$.

Dati due stati $Q_i = (M, St, E)$ e $Q'_i = (M', St', E')$, $Q_i \sqsubseteq Q'_i$ se e solo se $M \sqsubseteq M'$, $St \sqsubseteq St'$, $E \sqsubseteq E'$.

L'interprete astratto utilizza un *GlobalState* Q per l'esecuzione di un metodo. Se il metodo ha n istruzioni, Q è la sequenza degli stati Q_i, uno per ogni istruzione alla posizione i all'interno del codice CIL. Inoltre, Q include un stato speciale, chiamato Q_f, che non corrisponde ad alcuna istruzione, ma rappresenta lo stato finale, ottenuto dopo l'esecuzione dell'ultima istruzione del

metodo.

Il dominio dei *Global States* contiene uno stato speciale *error*. I *Global States* sono parzialmente ordinati in base alla relazione che c'è tra gli stati delle istruzioni: $Q \sqsubseteq Q'$, se e solo se $\forall i, Q_i \sqsubseteq Q'_i$. Inoltre, error è l'elemento al top sul dominio dei *Global States*: $Q \sqsubseteq error$ per ogni Q.

L'inteprete astratto si basa su un insieme di regole: viene definita una regola per ogni tipo di istruzione. L'interprete genera una catena di *Global States*$\{Q^j\}$, in cui ogni stato è ottenuto dal precedente in base all'applicazione delle regole. In § 5.6 sono riportate le regole per SIFDotNet. Nella stessa figura, si usa scrivere $Q_r := Q_r \sqcup (...)$. Questa simbologia indica che il global state Q è lo stesso tranne che per lo stato Q_r (corrispondente all'istruzione r), che è cambiato come specificato a destra dell'operatore :=

L'esecuzione astratta è definita partendo da un global state iniziale che riflette lo stato del Verificatore CIL all'entrata del metodo. Assumiamo che tutte le istruzioni dipendano dal nodo $START$ del CFG. $START$ vieni inizializzato con il livello della classe in cui è definito il metodo. Dato un metodo $C.mt$, il global state iniziale Q^0 è la sequenza degli stati iniziali ${Q_i}^0 = ({M_i}^0, {St_i}^0, {E_i}^0)$ di ogni istruzione i, come specificato nel seguito. Se $i \neq 0$, cioè $\beta[i]$ non è la prima istruzione:

- ${M_i}^0(x) = \sigma_0$ per ogni registro x
- ${St_i}^0 = \perp_{St}$ lo stack viene impostato al Bottom Element del dominio degli stacks.
- ${E_i}^0 = \{(START, S(C)), (c_0, \sigma_0)...(c_n, \sigma_0)\}$ l'ambiente è impostato al livello minimo per tutte le dipendenze, eccetto $START$.

Lo stato iniziale per la prima istruzione differisce dalla definizione precedente per l'inizializzazione dei registri x_0 e $x_1..x_{n+1}$, dell'ambiente e lo stack:

- ${M_0}^0(x_0) = S(C)\ x_0$ (contiene il riferimento all'oggetto) è inizializzato con il livello di sicurezza specificato per la classe cui l'oggetto appartiene.
- ${M_0}^0(x_1)..{M_0}^0(x_{n+1}) = P(C.mt)\ x_1..x_{n+1}$ (i parametri del metodo) sono inizializzati con i livelli di sicurezza impostati per i parametri di $C.mt$
- ${St_0}^0 = \lambda$ lo stack è vuoto
- ${E_0}^0 = \{(START, S(C)), (c_0, \sigma_0)...(c_n, \sigma_0)\}$ ${E_0}^0$ è settato come ${E_i}^0$.

Quando si applica la regola per la generica istruzione i, si procede come segue:

1. viene calcolato lo stato successivo di i, effettuando l'esecuzione astratta di i con lo stato Q_i. Per esempio, se viene eseguita l'istruzione `ldloc x`, e $Q_i = (M, St, E)$, lo stato successivo ad i ha memoria ed ambiente uguali a Q_i, mentre per quanto riguarda lo stack St, viene effettuato il *push* del Least Upper Bound tra il contenuto di x e l'ambiente E.
2. lo stato Q'_i, successivo ad i, viene unito (operazione di $merge$) con lo stato di tutte le istruzioni successive ad i. Sia j una istruzione successiva ad i. L'operazione di $merge$ è effettuata mediante il calcolo del Least Upper Bound tra il valore originale di Q_j e Q'_i. Per esempio, lo stato $after$ dell'istruzione `i: ldloc x` viene unito con lo stato Q_{i+1}.

Come conseguenza, se un'istruzione i ha molti predecessori, Q_i è il Least Upper Bound degli stati $after$ di tutte le istruzioni precedenti.

Le regole per l'istruzione if meritano maggiori chiarimenti. Quando viene eseguita l'istruzione `i:if j`: viene aggiornato l'ambiente delle due istruzioni successive ($i+1$ e j) perché tenga in considerazione il livello di sicurezza della condizione testata dall'istruzione condizionale.

La regola per `getfield C1.f` fa il *push* sullo stack del Lead Upper Bound tra il livello di sicurezza di $C1$ (che è anche il livello di sicurezza di $C1.f$), il livello di sicurezza del riferimento a $C1$ (che potrebbe essere differente di $S(C1)$) e l'ambiente.

5.5.1 Violazione del Secure Information Flow

Viene individuata una violazione del *Secure Information Flow* durante l'applicazione delle regole per la $putfield$, *invoke* e $return$:

- `putfield C1.f`

  Viene verificato che il livello del valore che verrà scritto nel campo $C1.f$ sia minore o uguale al livello di sicurezza della classe C1. Questo livello è il Least Upper Bound tra la locazione al top dello stack, il livello del riferimento a $C1$ e l'ambiente;

- `invoke C1.mt1`

  Viene verificato che il Least Upper Bound tra il livello dei parametri (in cima allo stack) e l'ambiente sia minore o uguale a $P(C1.mt1)$ e che il Least Upper Bound tra il livello del riferimento e l'ambiente sia minore o uguale a $S(C1)$. Il valore di ritorno di $C1.mt1$ viene inserito in cima allo stack dello stato $after$. Esso è il Least Upper Bound tra il livello del riferimento, l'ambiente ed il livello specificato da $R(C1.mt1)$;

- `return`

  Viene verificato che il Least Upper Bound tra il livello del valore di ritorno (in cima allo stack) e l'ambiente sia minore o uguale al livello specificato da $R(C.mt)$.

Assumiamo che, se viene individuato un errore, il corrispondente Global State sia impostato ad *error*. Questo significa che è necessario definire un regola addizionale per le tre istruzioni $putfield$, *invoke*, *return* (le regole sono indicate in § 5.6).

5.6 Regole di semantica astratta

pop

$\beta[i] = \texttt{pop}, \quad Q_i = (M, k \cdot S, E)$
$Q_{i+1} \sqcup = (M, S, E)$

dup

$\beta[i] = \texttt{dup} \quad Q_i = (M, k \cdot S, E)$
$Q_{i+1} \sqcup = (M, (level(E) \sqcup k) \cdot (level(E) \sqcup k) \cdot S, E)$

op

$\beta[i] = \texttt{op} \quad Q_i = (M, k_1 \cdot k_2 \cdot S, E)$
$Q_{i+1} \sqcup = (M, (level(E) \sqcup k_1 \sqcup k_2) \cdot S, E)$

load

$\beta[i] = \texttt{load x} \quad Q_i = (M, S, E)$
$Q_{i+1} \sqcup = (M, (level(E) \sqcup M(x)) \cdot S, E)$

store

$\beta[i] = \texttt{store x} \quad Q_i = (M, k \cdot S, E)$
$Q_{i+1}\sqcup = (M[(level(E) \sqcup k)/\texttt{x}], S, E)$

const

$\beta[i] = \texttt{const}\, d \quad Q_i = (M, S, E)$
$Q_{i+1}\sqcup = (M, level(E) \cdot S, E)$

if*cond*

$\beta[i] = \texttt{if}cond \quad j, \quad Q_i = (M, k \cdot S, E)$
$Q_{i+1}\sqcup = (M, S, E \sqcup (i, k));\ Q_j\sqcup = (M, S, E \sqcup (i, k))$

goto

$\beta[i] = \texttt{goto j}$
$Q_j\sqcup = Q_i$

new

$\beta[i] = \texttt{new A}, Q_i = (M, S, E)$
$Q_{i+1}\sqcup = (M, (level(E) \sqcup S(A)) \cdot S, E)$

ldfld

$\beta[i] = \texttt{ldfld C1.f} \quad Q_i = (M, r \cdot S, E)$
$Q_{i+1}\sqcup = (M, (\mathcal{S}(C1) \sqcup level(E) \sqcup r) \cdot S, E)$

stfld

$\beta[i] = \texttt{stfld C1.f}$ $Q_i = (M, k \cdot r \cdot S, E), \quad k \sqcup r \sqcup level(E) \sqsubseteq \mathcal{S}(C1)$
$Q_{i+1}\sqcup = (M, S, E)$

invoke

$\beta[i] = \texttt{invoke C1.mt}$

$Q_i = (M, k \cdot r \cdot S, E) \quad k \sqcup level(E) \sqsubseteq \mathcal{P}_{C1.mt_1}, r \sqcup level(E) \sqsubseteq \mathcal{P}_{C1.mt_0}$

$Q_{i+1}\sqcup = (M, (\mathcal{R}_{C1.mt} \sqcup level(E) \sqcup r) \cdot S, E)$

return

$\beta[i] = \alpha\texttt{return} \quad Q_i = (M, k \cdot S, E) \quad (k \sqcup level(E)) \sqsubseteq \mathcal{R}_{C.mt}$

$Q_f\sqcup = (M, S, E)$

stfld_err

$\beta[i] = \texttt{stfld C1.f} \quad Q_i = (M, k \cdot r \cdot S, E), \quad k \sqcup r \sqcup level(E) \not\sqsubseteq \mathcal{S}(C1)$

$Q := \texttt{error}$

invoke_err

$\beta[i] = \texttt{invoke C1.mt1}$

$Q_i = (M, k \cdot r \cdot S, E) \quad k \sqcup level(E) \not\sqsubseteq \mathcal{P}_{C1.mt_1} \; or \; r \sqcup level(E) \not\sqsubseteq \mathcal{P}_{C1.mt_0}$

$Q := \texttt{error}$

return_err

$\beta[i] = \alpha\texttt{return} \quad Q_i = (M, k \cdot S, E) \quad k \sqcup level(E) \not\sqsubseteq \mathcal{R}_{C.mt}$

$Q := \texttt{error}$

Parte III

Specifiche di progetto ed implementazione di SIFDotNet

Capitolo 6

SIFDotNet: un tool per la tutela delle informazioni private

6.1 Introduzione

SIFDotNet è uno tool di analisi statica del flusso di informazioni sicuro per Microsoft .NET: è stato progettato e sviluppato ex novo secondo quando indicato in § 5, utilizzando un approccio per la verifica della sicurezza delle informazioni sviluppato in [1] e le esperienze maturate nell'ambito dell'analisi statica del codice in ambiente Java.

Lo strumento realizzato come illustrato nella Figura 6.1 è dotato delle seguenti funzionalità:

1. Disassembla un qualunque Assembly .NET, in particolare è in grado di disassemblare file eseguibili e librerie .dll scritti con un qualunque linguaggio .NET (J#, VB.NET, C#, Cobol, Managed C++, etc..)

2. Fornisce un **Report** dettagliato della struttura dell'Assembly .NET disassemblato, descrivendo le informazioni in formato XML

3. Effettua l'analisi con il metodo del Secure Information Flow, testando ogni metodo, dichiarato all'interno di un Tipo, alla ricerca di una violazione della riservatezza delle informazioni.

4. Rileva violazioni della riservatezza delle informazioni, fornendo un rapporto dettagliato delle istruzioni utilizzate per carpire i dati personali.

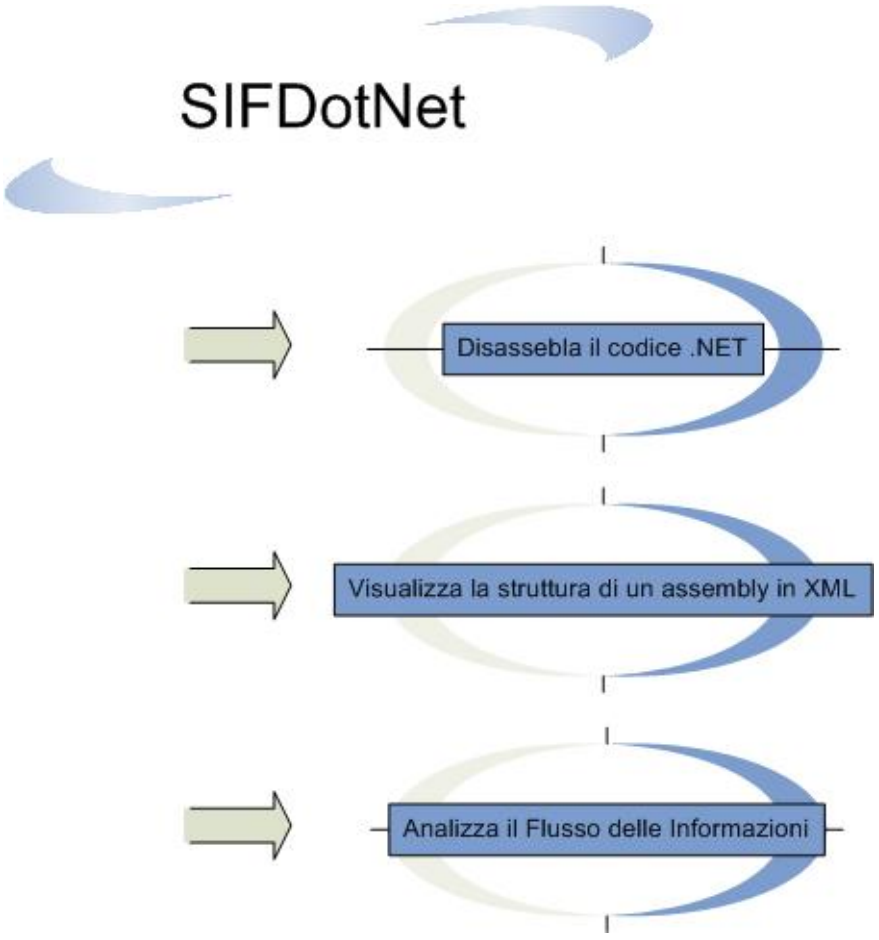


Figura 6.1: Le caratteristiche principali di SIFDotNet

6.2 Requisiti di progetto

SIFDotNet è stato progettato per rispondere a 5 requisiti fondamentali (Figura 6.2):

1. Prendere in input un Assembly .NET e disassemblarlo.
2. Fornire all'utente la possibilità di analizzare la struttura dell'Assembly
3. Permettere una agevole impostazione del livello di segretezza delle informazioni
4. Individuare tentativi di violazione della privacy
5. Modularità ed espandibilità

6.2.1 Input e MSIL

Un Assembly .NET è, essenzialmente, strutturato come in Figura 1.4. Non è semplice reperire informazioni sul formato di un file .NET: l'unica fonte disponibile risulta essere l'ECMA CLI specification [3], in cui vengono fornite grandi quantità di informazioni per un totale di circa 500 pagine.

Durante la fase di *"Studio di fattibilità"* e *"Analisi dei Requisiti"* ci siamo posti il problema di come implementare il sottosistema dedicato alla lettura di un Assembly. L'operazione richiede numerose conoscenze sull'architettura .NET ed uno studio dettagliato della documentazione ECMA. Inoltre, il problema più grosso era costituito dal fatto che l'implementazione del sottosistema avrebbe richiesto molto tempo e, se non realizzata in modo completo, avrebbe richiesto un continuo lavoro di aggiornamento ed una grossa perdita di tempo lavoro.

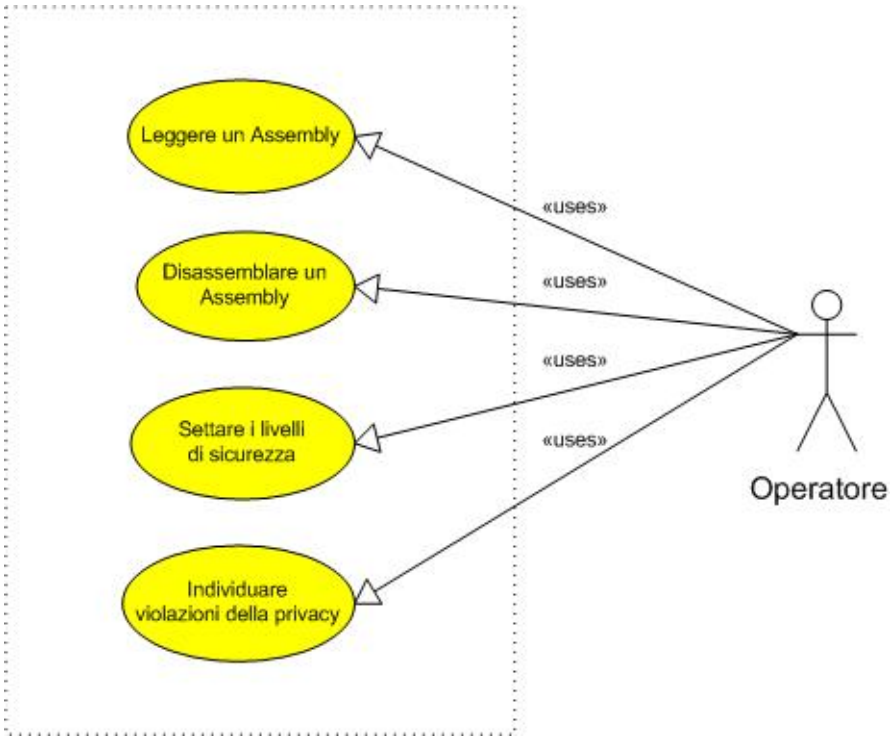


Figura 6.2: Funzionalità necessarie a SIFDotNet

Abbiamo, così, rivolto la nostra attenzione al mondo Open-Source, in particolare ci siamo interessati al giovane progetto MONO[1] sponsorizzato da Novell per creare un'implementazione gratuita di .NET e, soprattutto, portabile su altre piattaforme. Il grosso vantaggio di un progetto di questo tipo è derivato dal fatto che tutto il codice sviluppato è disponibile per una consultazione e quindi ricco di validi suggerimenti. In questo ambito non è stato difficile accorgersi che le nostre difficoltà e perplessità erano diffuse, tuttavia la soluzione al problema era fornita da una libreria open-source sviluppata da Lutz Roeder[2] (ILReader.dll).

È stato quindi, subito, chiaro che nella fase di progettazione e sviluppo del prototipo la libreria *ILReader.dll* costituiva la soluzione migliore, limitando tutto il lavoro di sviluppo del sottosistema ad un'operazione di interfacciamento della libreria con il resto dei sottosistemi.

Questo primo requisito, quindi, è stato risolto implementando, come spiegato dettagliatamente nel seguito, una serie di classi di supporto atte a fare da interfaccia con la libreria ed a separare l'implementazione del sottosistema dagli altri.

6.2.2 Semplice consultazione della struttura di un Assembly

Questo requisito fondamentale era necessario per permettere, soprattutto nella fase di prototipazione, una veloce ispezione dell'Assembly. L'analisi statica

[1] http://www.go-mono.com/

[2] Maggiori informazioni su Lutz Roeder possono essere reperite presso il sito di riferimento

per il flusso sicuro di informazioni richiede la conoscenza della struttura di un Assembly.

In questo caso le scelte sono state guidate da altri requisiti, molto importanti:

- le informazioni riguardanti la struttura di un Assembly devono poter essere consultate in vari modi:
  - un operatore ha la necessità di leggere le informazioni per impostare il livello di segretezza dei dati
  - il programma ha la necessità di consultare le informazioni per recuperare eventuali informazioni di sicurezza sull'Assembly
- il formato delle informazioni deve essere comprensibile in modo semplice dall'operatore, ma, anche, in modo veloce dal software di analisi

La scelta si è, così, rivolta verso la creazione di un sottosistema dedicato alla gestione di queste informazioni, dotato di una semplice interfaccia nei confronti del sottosistema di analisi e di una interfaccia user-friendly nei confronti dell'operatore. Il sottosistema memorizza le informazioni in formato XML, come specificato nel seguito, in maniera tale che l'operatore lo possa consultare con l'aiuto di un qualunque browser.

6.2.3 Un modo semplice per impostare i livelli di segretezza

Con la scelta del precedente requisito, impostare i livelli di segretezza delle informazioni utilizzate dall'Assembly diventa un'operazione agevole ed espandibile.

L'utilizzo del formato XML permette numerose soluzioni per l'*editing* dei livelli di segretezza. Questi possono essere inseriti modificando manualmente il file (scelta preferita nella fase di prototipazione), oppure fornendo una semplice interfaccia grafica che espone in modo chiaro e funzionale la struttura dell'Assembly e permette la modifica dei livelli di segretezza.

6.2.4 Individuare violazioni della Privacy

Si tratta del requisito principale, SIFDotNet deve essere in grado di rilevare violazioni della segretezza delle informazioni. Il rispetto di questo principio è garantito dalla tecnica descritta nel Capitolo 5. Alle informazioni viene assegnato un livello di sicurezza e la presenza di un flusso di informazioni illecito può essere rilevato richiedendo che le informazioni con un determinato livello non fluiscano a livelli inferiori. Dato un programma in cui a ciascuna variabile è assegnato un livello di sicurezza, esso gode della proprietà di flusso sicuro di informazioni se, alla fine della sua esecuzione, il valore di ogni variabile non dipende dal valore iniziale delle variabili con livello di sicurezza superiore.

Per garantire lo sviluppo di una soluzione modulare ed espandibile il compito di effettuare l'analisi statica del codice è stato assegnato ad un apposito sottosistema, in grado di comunicare con il disassemblatore ed il gestore dei livelli di sicurezza esclusivamente mediante le interfacce fornite da questi ultimi.

6.2.5 Modularità ed espandibilità

Dall'analisi dei precedenti requisiti e le soluzioni proposte, la modularità ed espandibilità risulta essere una proprietà implicita ed esente di ulteriori accor-

gimenti.

I sottosistemi sono completamente disgiunti, questo garantisce una modifica degli stessi o riprogettazione nel completo rispetto del funzionamento del resto del programma. In futuro il disassemblatore potrà essere sviluppato ex-novo, o modificato, in seguito ad un aggiornamento delle specifiche .NET ed il gestore dei livelli di sicurezza potrà essere ampliato per aggiungere il supporto di nuove funzionalità.

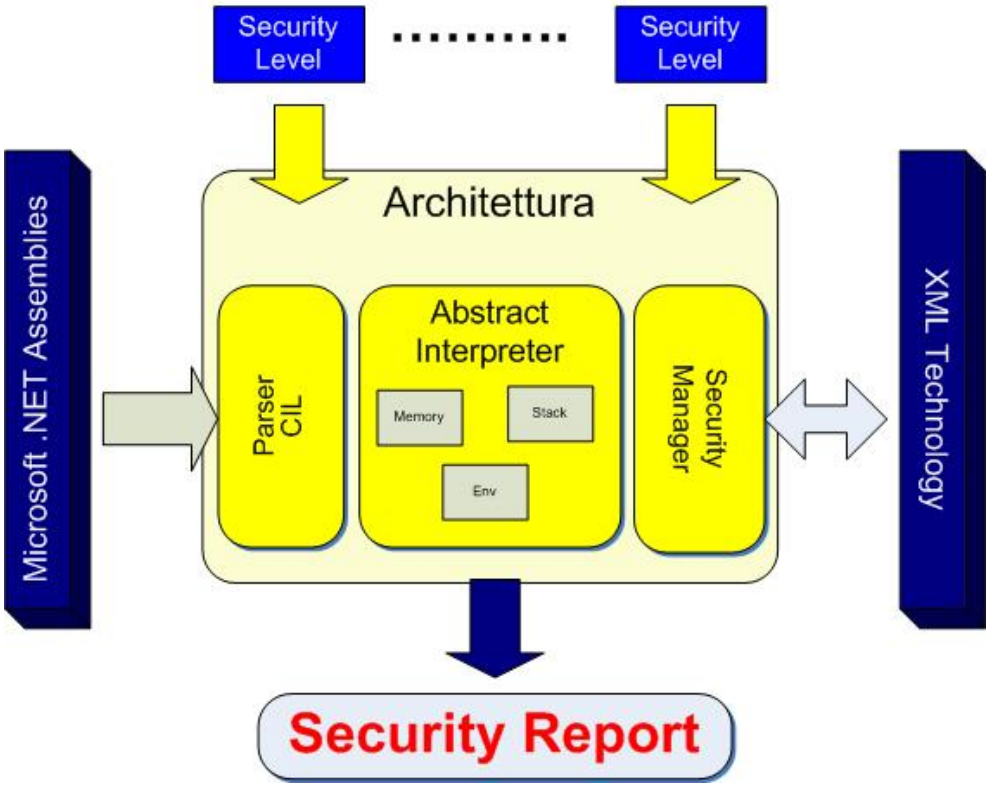


Figura 6.3: Tre sottosistemi indipendenti

6.3 L'architettura di alto livello

SIFDotNet, come illustrato in Figura 6.3, si compone di tre sottosistemi:

1. Sottosistema "Parser"
2. Sottosistema "Abstract Interpreter"
3. Sottosistema "Security Manager"

6.3.1 Il Parser

Lo scopo del Parser è quello di fare da interfaccia tra "Abstract Interpreter" e l'assembly .NET, il principio di funzionamento è illustato in Figura 6.4.

Il parser riceve in input un Assembly .NET e, su richiesta, è in grado di fornire il codice assembly di un particolare metodo. Il suo compito è quello di inglobare tutta la logica necessaria per disassemblare un file ed agevolare, quindi, il compito degli altri sottosistemi.

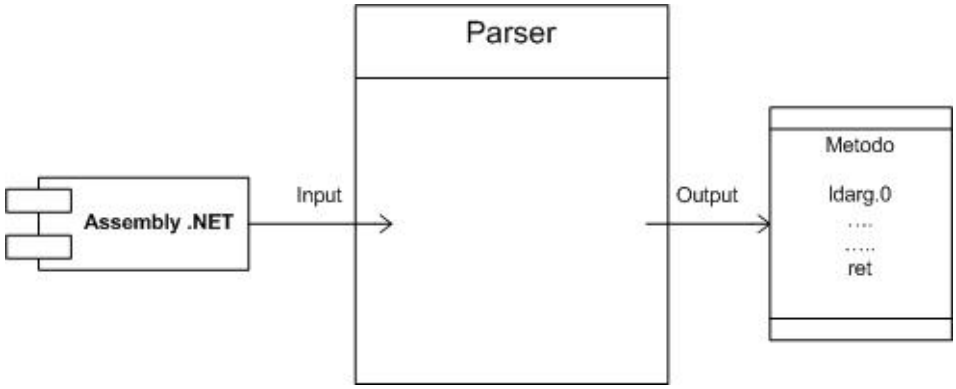


Figura 6.4: Il parser fornisce il codice CIL di un Metodo

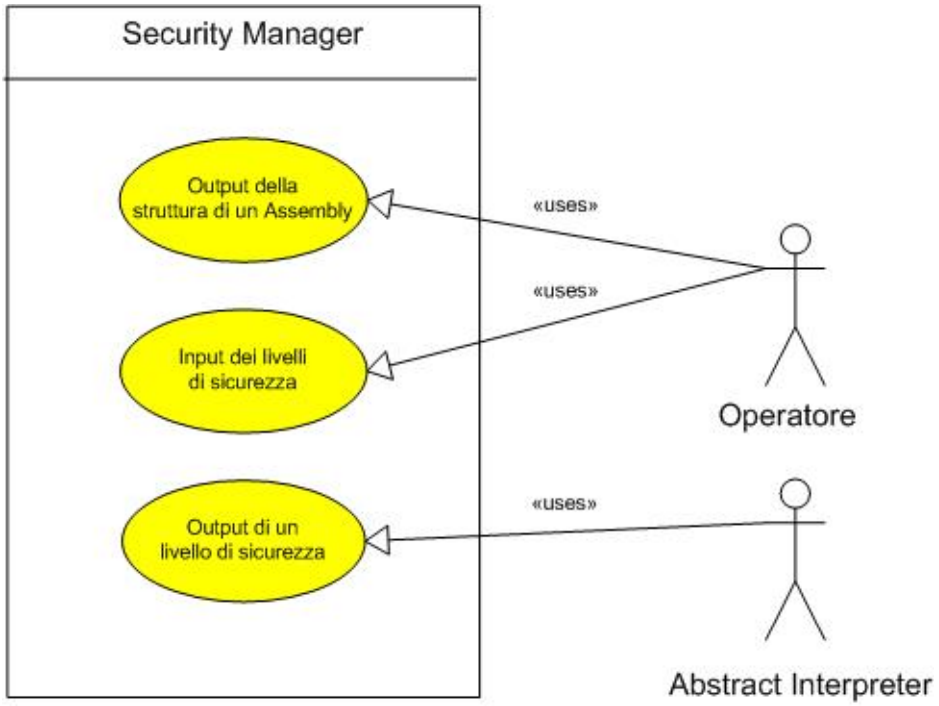


Figura 6.5: Casi d'uso del sottosistema

6.3.2 Security Manager

Il sottosistema Security Manager, Figura 6.5, svolge varie funzioni:

1. Effettua l'output su file della struttura di un Assembly
2. Si occupa di gestire l'input/output dei livelli di sicurezza con l'operatore
3. Fornisce al sottosistema Abstract Iterpreter il livello di sicurezza dell'elemento richiesto

Il suo compito richiede la capacità di analizzare la struttura di un Assembly, quindi i Tipi dichiarati ed i rispettivi metodi, con argomenti e variabili locali. Questa funzionalità non richiede la conoscenza del codice CIL, per questo mo-

tivo le chiamate al sottosistema Parser sono molto limitate e la loro funzione è spiegata in seguito.

6.3.3 Abstract Interpreter

Il sottosistema Abstract Interpreter rappresenta il *cuore* di SIFDotNet. Suo compito è analizzare l'assembly fornito dall'operatore, metodo per metodo, alla ricerca di una violazione della segretezza delle informazioni, come illustrato in Figura 6.6. L'interprete astratto ha anche il compito di gestire l'output delle informazioni con l'operatore, per comunicare eventuali violazioni e fornire indicazioni circa il metodo e l'istruzione che ha tentato di manipolare in modo scorretto dati ad alto livello di segretezza.

Il sottosistema si interfaccia con il *Parser* per richiedere il codice CIL di un metodo ed con il *Security Manager* per richiedere il livello di sicurezza di un Tipo.

6.4 Dettagli sull'architettura ed implementazione

SIFDotNet, come descritto in § 6.3, è composto da tre sottosistemi logici:

1. Parser
2. Security Manager
3. Abstract Interpreter

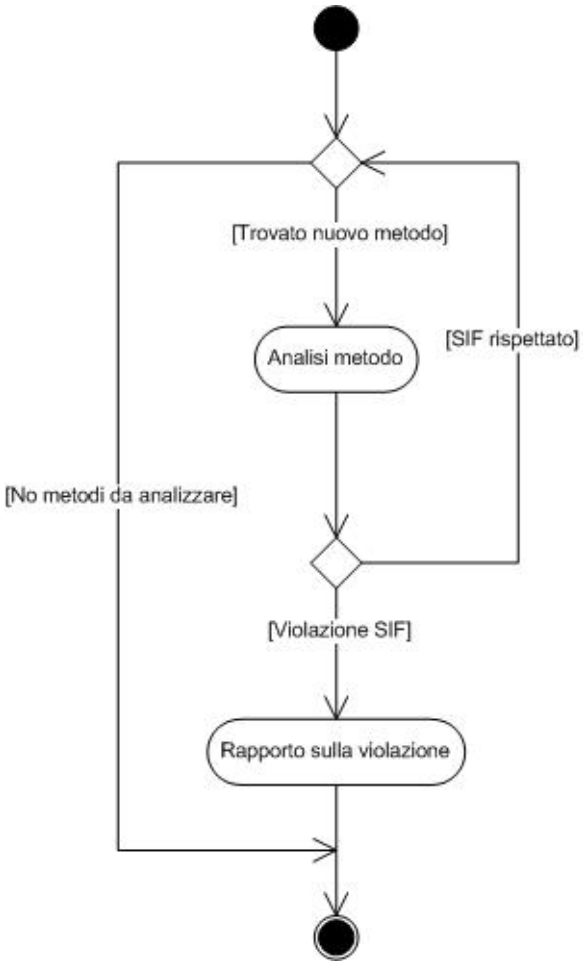


Figura 6.6: Principio di funzionamento dell'interprete astratto

Questi sottosistemi trovano una corrispondenza uno a uno con i NameSpace implementati, che sono stati nominati rispettivamente:

1. ParserIL
2. SecurityManager
3. InterpreteIL

In base alle specifiche di progetto i tre sottosistemi risultano completamente indipendenti ed, eventualmente, riutilizzabili separatamente per scopi futuri.

Ci occuperemo ora di analizzare in dettaglio i sottosistemi, partendo dai due sottosistemi di utilità, *Parser e Security Manager* per poi entrare nel dettaglio dell'*Abstract Interpreter* che costituisce il cuore di tutta l'applicazione.

6.4.1 ParserIL

ParserIL è il sottosistema che si occupa di disassemblare un Assembly .NET e fornire servizi agli altri sottosistemi. Come spiegato in § 6.2.1, esso utilizza la libreria *ILReader.dll* per ottenere tutte le informazioni necessarie da un Assembly. In Figura 6.7 è illustrata l'architettura del ParserIL. Il Tipo *Parser* rappresenta il cuore del sottosistema, esso rende disponibili tre metodi pubblici con i quali è possibile effettuare delle richieste di servizio (Figura 6.8):

- *ILMethodBody GetMethodBody(Type t, MethodInfo m)* utilizzato per richiedere al Parser il codice CIL di un metodo m, appartenente ad tipo t. Il tipo restituito, infatti, come sarà chiaro più avanti, contiene tutte le informazioni del metodo disassemblato.

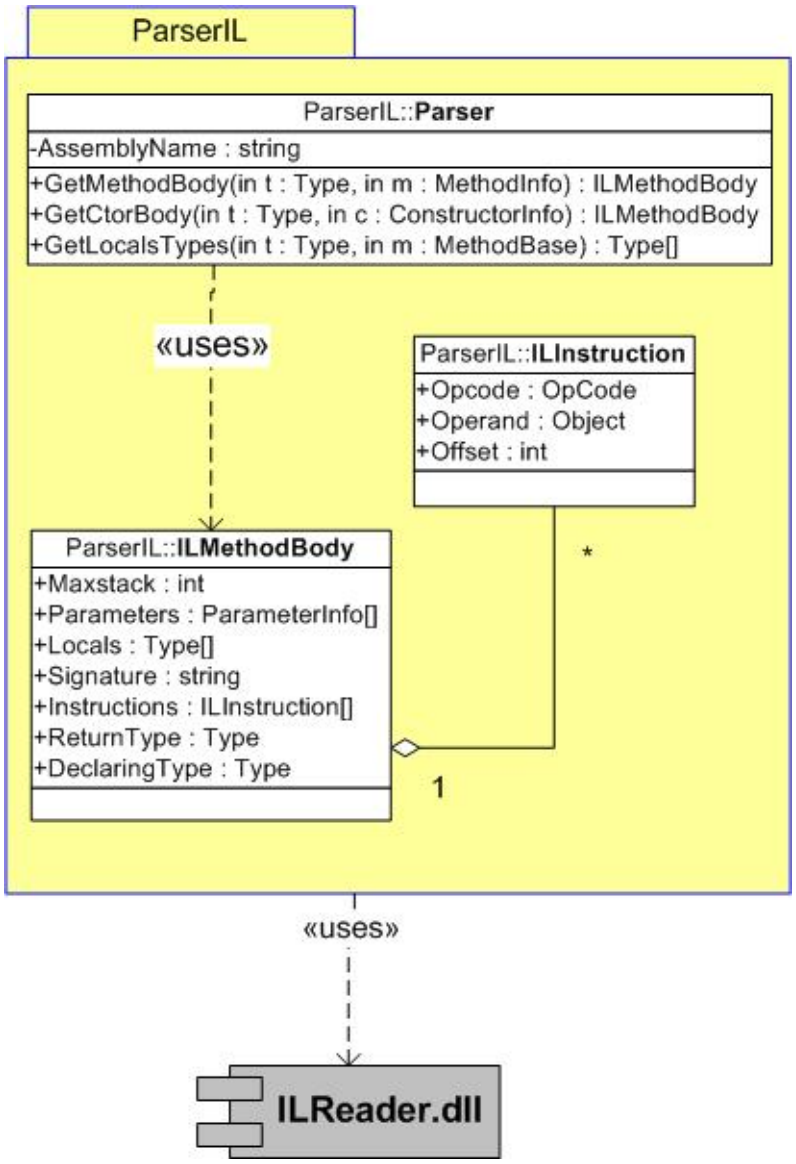


Figura 6.7: ParserIL: sono visibili i Tipi esportati

- *ILMethodBody GetCtorBody(Type t, ConstructorInfo c)* utilizzato per richiedere il codice CIL di un costruttore c del tipo t.

- *Type[] GetLocalsTypes(Type t, MethodBase m)* utilizzato se sono necessarie soltanto le variabili locali di un metodo m, appartenente al tipo t.

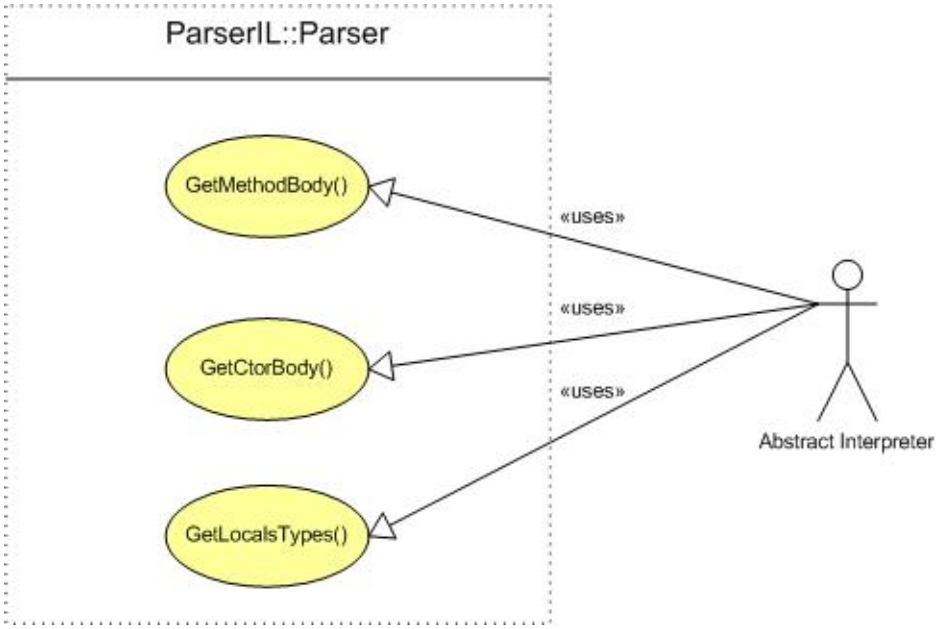


Figura 6.8: Utilizzo del Tipo ParserIL::Parser

I due metodi *GetMethodBody e GetCtorBody* restituiscono un Tipo di nome *ILMethodBody*, questo rappresenta la struttura di un metodo CIL che viene rappresentata in questo modo:

- *MaxStack*: Dimensione massima dell'evaluation stack

- *Parameters*: Array di Tipi che rappresentano gli argomenti del metodo

- *Locals*: Array di Tipi che rappresentano le variabili locali

- *Signature*: Firma del metodo
- *Instructions*: Array di ILInstruction che rappresenta l'insieme di istruzioni CIL del metodo
- *ReturnType*: Tipo ritornato dal metodo
- *DeclaringType*: Tipo proprietario del metodo

Un'istruzione CIL viene rappresentata mediante il Tipo *ILInstruction*, il quale esporta tre informazioni:

1. *Opcode*: Opcode dell'istruzione CIL
2. *Operand*: Operando
3. *Offset*: Offset dell'istruzione all'interno del metodo

Per quanto riguarda l'utilizzo da parte del **ParserIL** della libreria ILReader.dll in Figura 6.9 si descrivono le operazioni necessarie per portare a termine la richiesta di un metodo da parte di uno *User*. La libreria mette a disposizione il Tipo *ModuleReader* per effettuare le operazioni di lettura dell'assembly. Utilizzando il metodo *MethodBody.GetMethodBody()* è possibile recuperare il codice CIL del metodo richiesto.

6.5 SecurityManager

SecurityManager è il sottosistema adibito alla gestione ed assegnamento dei livelli di sicurezza. Per svolgere queste mansioni il sistema si compone come

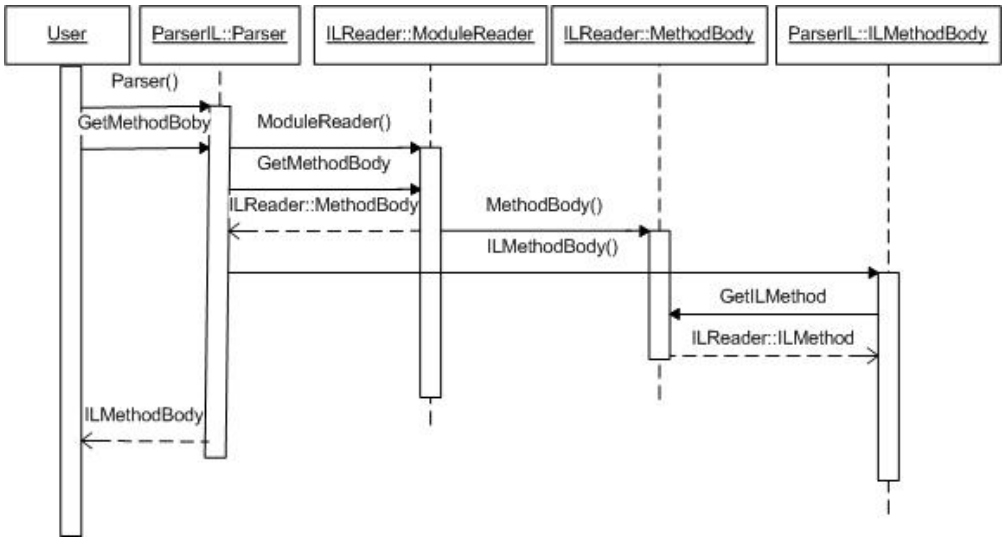


Figura 6.9: ParserIL: Sequenza delle azioni intraprese in seguito ad una richiesta di servizio

rappresentato in Figura 6.10. In Figura 6.11 sono, invece, illustrati i servizi offerti.

I Tipo *SecWriter* deve essere istanziato per richiedere al **SecurityManager** di effettuare il *Dump* dell'Assembly (operazione *DumpAssemblySchema*).

L'operazione di Dump di un Assembly necessita di ulteriori chiarimenti; come sappiamo un Assembly è costituito da un insieme di Metodi facenti parte di un certo numero di Tipi e da un insieme di NameSpace, che raggruppano logicamente i Tipi. L'operazione di *Dump* scrive su file la struttura dell'Assembly, utilizzando, per la descrizione delle informazioni, una semantica di tipo XML (esempio Figura 6.12)

SecWriter è in grado di ricavare la struttura di un Assembly senza ricorrere al *Parser*, o dover conoscere i dettagli del codice CIL. Il sottosistema SecurityManager, come anche gli altri sottosistemi, infatti, fanno largo uso della **Reflection**. La *Reflection* è la capacità di "riflettere" sulle proprietà di un

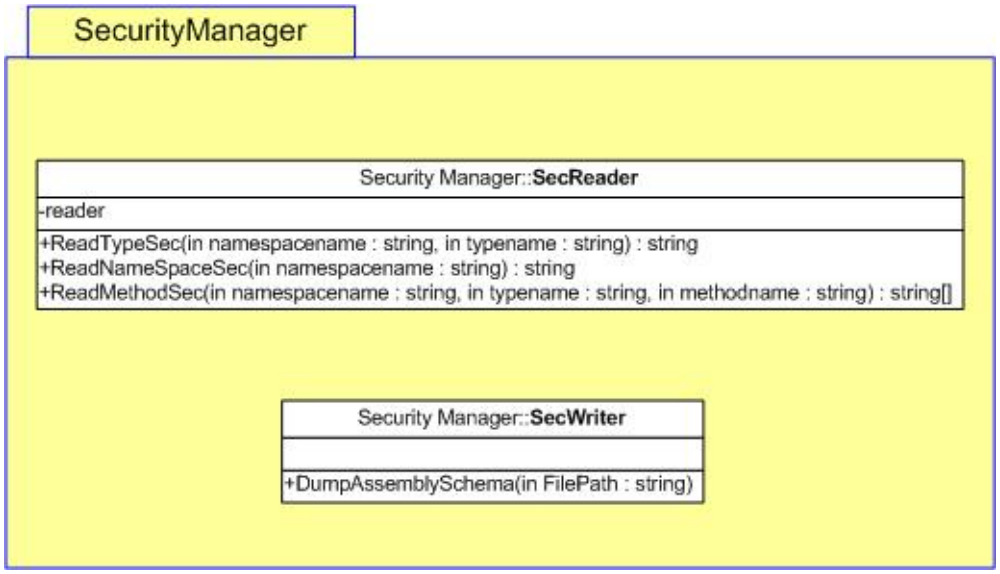


Figura 6.10: SecurityManager: due tipi pubblici per leggere e scrivere livelli di sicurezza

altro Assembly o di esso stesso. In pratica utilizzando la libreria di Tipi **System.Reflection** è possibile sapere quali Tipi sono dichiarati all'interno di un altro Assembly, quali i metodi, i parametri etc.. Un esempio di uso di questa tecnologia è stato già usato precedentemente, i Tipi, infatti, *MethodBase*, *MethodInfo*, *ParameterInfo* e *ConstructorInfo* sono stati usati per l'implementazione del **Parser**.

Il Tipo *SecWriter*, all'interno del metodo *DumpAssemblySchema*, utilizza la **Reflection** per ricavare la struttura dell'Assembly e poi fa uso del Tipo interno *XMLSecWriter* per scrivere su file le informazioni. In Figura 6.13 è disponibile qualche dettaglio sull'algoritmo utilizzato ed in Figura 6.14 si fa vedere l'interfaccia del Tipo XMLSecWriter.

SecurityManager fornisce anche un interfaccia per la lettura dei livelli di sicurezza. Il file XML, risultato dell'operazione di Dump, prevede, infatti, da

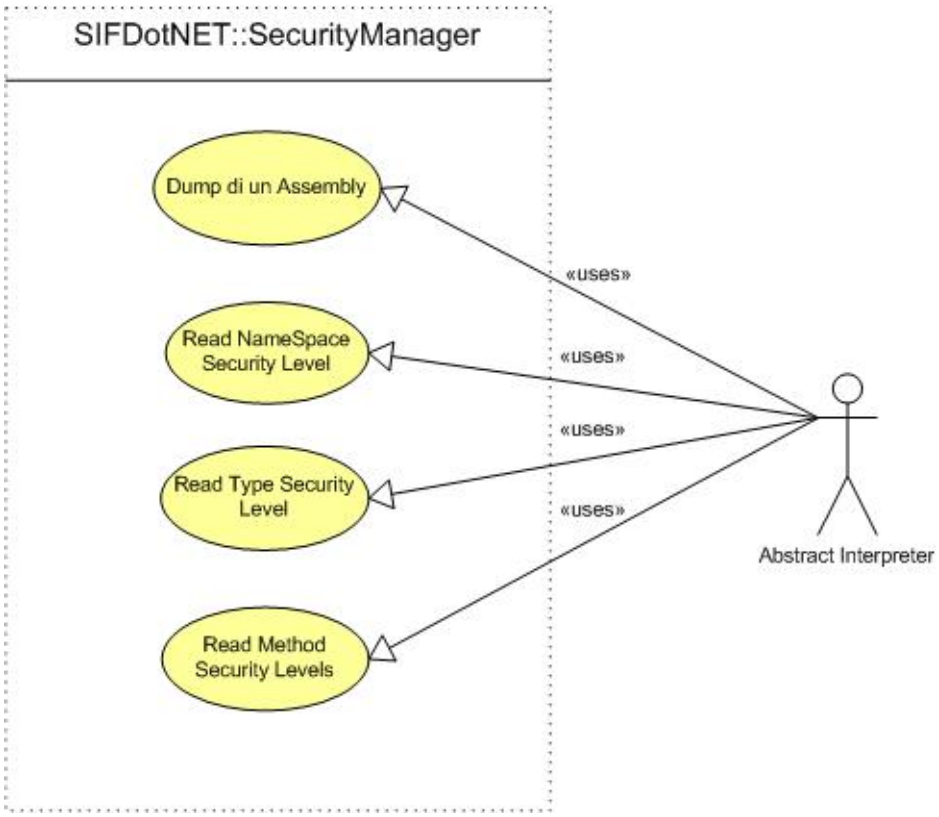


Figura 6.11: Servizi offerti dal SecurityManager

```
<Security_Settings>
  <Referenced_NameSpaces>
    <Namespace Name="System.Net" Security_Level="" />
    <Namespace Name="Project13" Security_Level="" />
    <Namespace Name="System.IO" Security_Level="" />
    <Namespace Name="System" Security_Level="" />
  </Referenced_NameSpaces>
  <Type Name="prova" NameSpace="Project13" Security_Level="">
    <Method Name="Void .ctor()" />
  </Type>
  <Type Name="SendFile" NameSpace="Project13" Security_Level="">
    <Method Name="Void .ctor()" />
    <Method Name="Void NoArgs()">
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void Send(System.String)">
      <Parameter Name="message" Type="System.String" Security_Level="Hight" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
  </Type>
  <Type Name="Class" NameSpace="Project13" Security_Level="">
    <Method Name="Void .ctor()" />
    <Method Name="Void Main(System.String[])">
      <Parameter Name="args" Type="System.String[]" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
  </Type>
</Security_Settings>
```

Figura 6.12: SecurityManager: Esempio di file prodotto chiamando il metodo *DumpAssemblySchema*

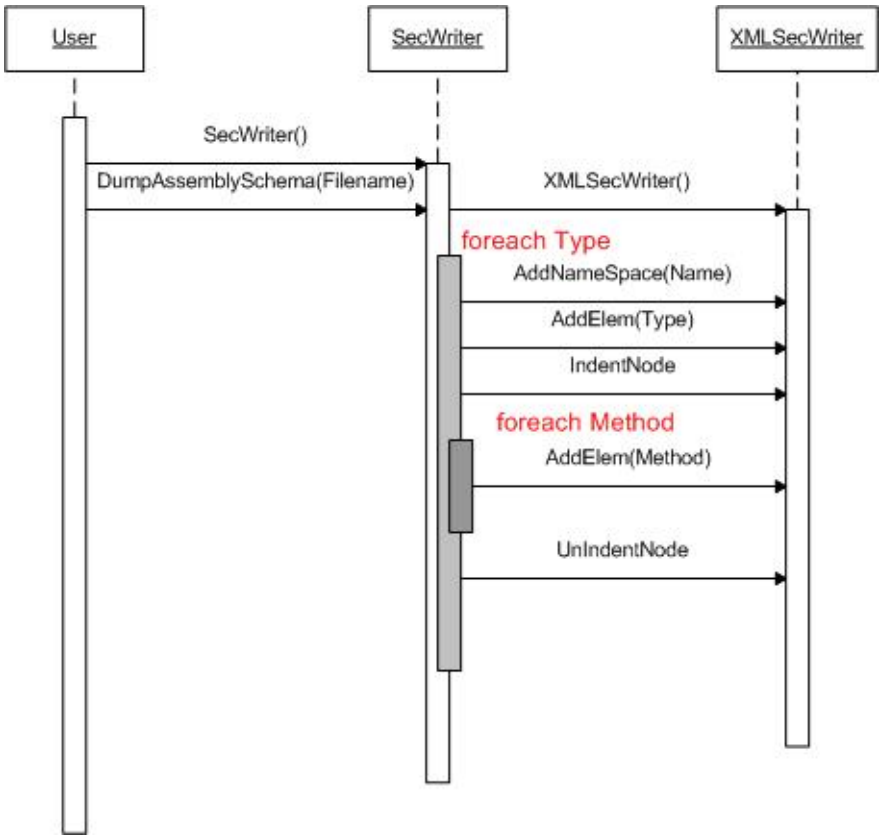
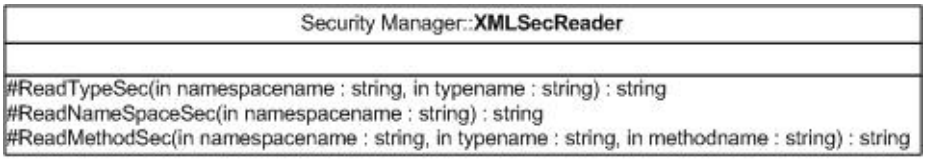


Figura 6.13: SecWriter: algoritmo



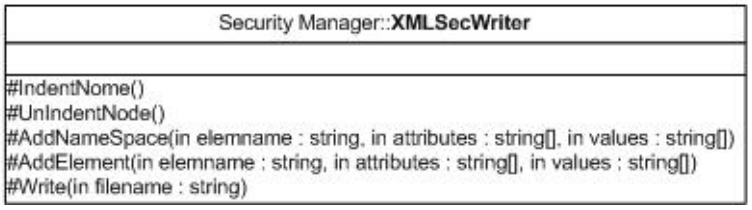


Figura 6.14: SecurityManager: interfacce dei Tipi Interni XMLSecWriter e XMLSecReader

parte dell'operatore, l'inserimento dei livelli di sicurezza per i Namespace, i Tipi, gli Argomenti ed il valore di ritorno dei Metodi. Come illustrato in Figura 6.10, il Tipo SecReader deve essere istanziato per la lettura dei livelli di sicurezza, esso prevede tre metodi:

1. *ReadNameSpaceSec*: questo metodo richiede il nome del Namespace e restituisce il rispettivo livello di sicurezza

2. *ReadTypeSec*: questo metodo richiede il nome del Tipo e del NameSpace proprietario del Tipo, è possibile, infatti, che Tipi con lo stesso nome si trovino in Namespace differenti. Il metodo restituisce il livello di sicurezza del Tipo.

3. *ReadMethodSec*: questo metodo richiede il nome del Tipo, del NameSpace e la firma del Metodo. Sono richieste tutte queste informazioni per individuare correttamente il Tipo proprietario del metodo e per rilevare il metodo giusto all'interno di un set di metodi definiti in Overloading. Il metodo restituisce un array di $n + 1$ livelli di sicurezza, corrispondenti agli n argomenti del metodo ed al valore di ritorno.

Il Tipo SecReader utilizza il Tipo XMLSecReader per delegare il compito di interpretare il file XML.

6.6 InterpreteIL

InterpreteIL è il sottosistema dedicato all'analisi statica del codice CIL secondo la teoria del Secure Information Flow, destritta in § 5. In Figura 6.15 possiamo osservare una visione d'insieme del sistema.

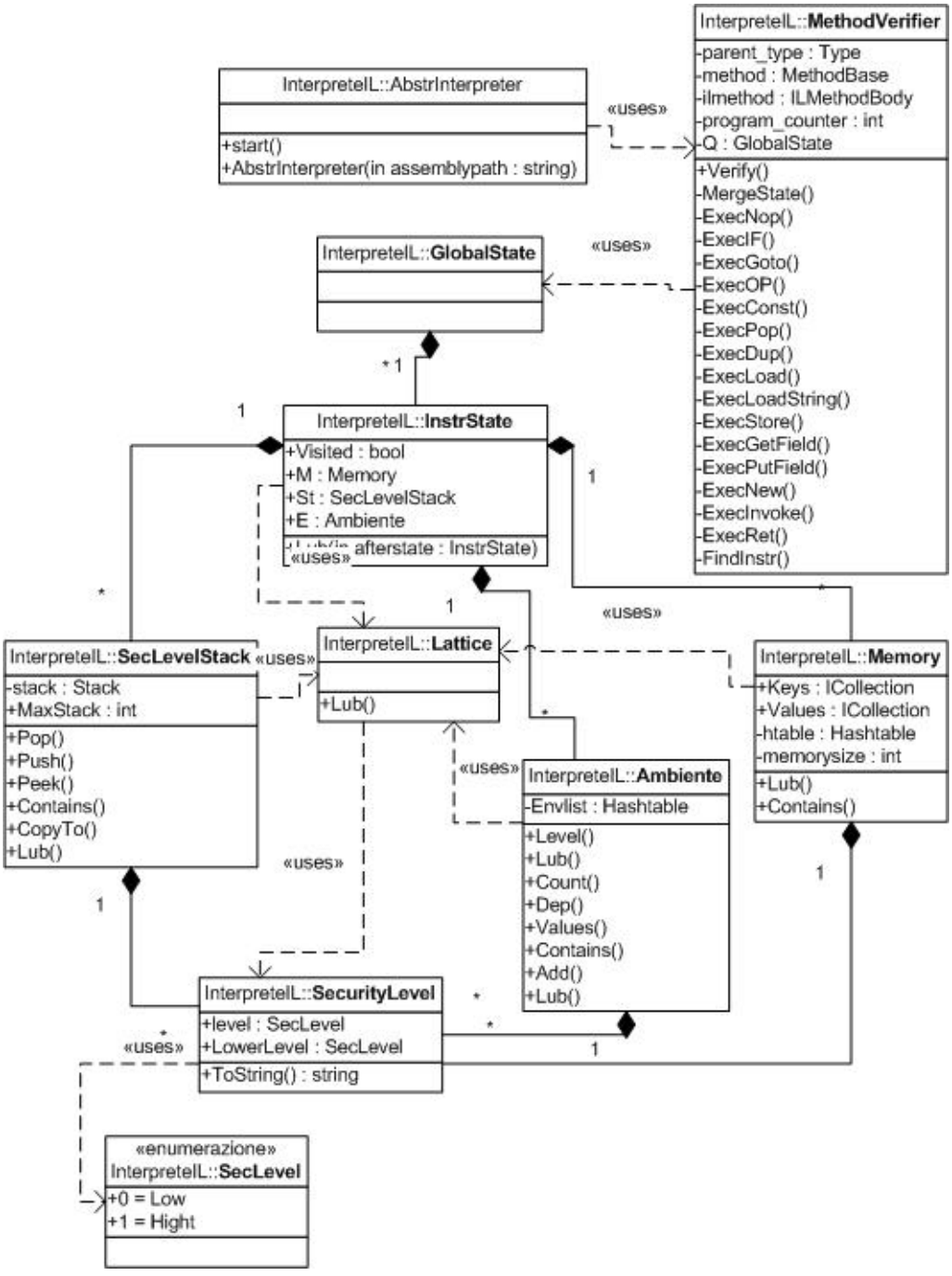


Figura 6.15: Schema complessivo

6.6.1 AbstrInterpreter

AbstrInterpreter si occupa della gestione dell'algoritmo di analisi statica del codice, coadiuvato dal Tipo *MethodVerifier* di cui parleremo in seguito. Prende in ingresso il *path* dell'Assembly .NET da analizzare e, utilizzando la **Reflection**, ricava la struttura dei Tipi, dichiarati nell'Assembly e di tutti i suoi metodi. Questa tecnica è molto semplice e poco invasiva perché non richiede la lettura del codice CIL ed è eseguibile in modo veloce con poche istruzioni.

AbstrInterpreter, per ogni Tipo dichiarato nell'Assembly, recupera tutti i costruttori ed i metodi, delega quindi la verifica del rispettivo codice CIL al *MethodVerifier*, il quale, una volta terminato, restituisce un'indicazione dell'esito dell'analisi. In conseguenza al risultato dell'analisi, *AbstrInterpreter* decide se continuare il test degli altri metodi, l'analisi precedente è andata a buon fine, o se interfacciarsi, prima, con l'operatore per indicare una violazione del Secure Information Flow. Prima di terminare, l'analisi viene comunque estesa a tutto l'Assembly, in modo tale da avere un idea chiara dei metodi che violano la privacy.

In Figura 6.16 è indicato l'algoritmo utilizzato.

6.6.2 MethodVerifier

Questo Tipo implementa il metodo per la verifica del Secure Information Flow come spiegato in § 5. Per questo scopo, è suo compito interpellare il sottosistema *ParserIL* per richiedere il codice CIL del metodo da analizzare ed il sottosistema *SecurityManager* per richiedere i livelli di sicurezza di tutti i Tipi incontrati,

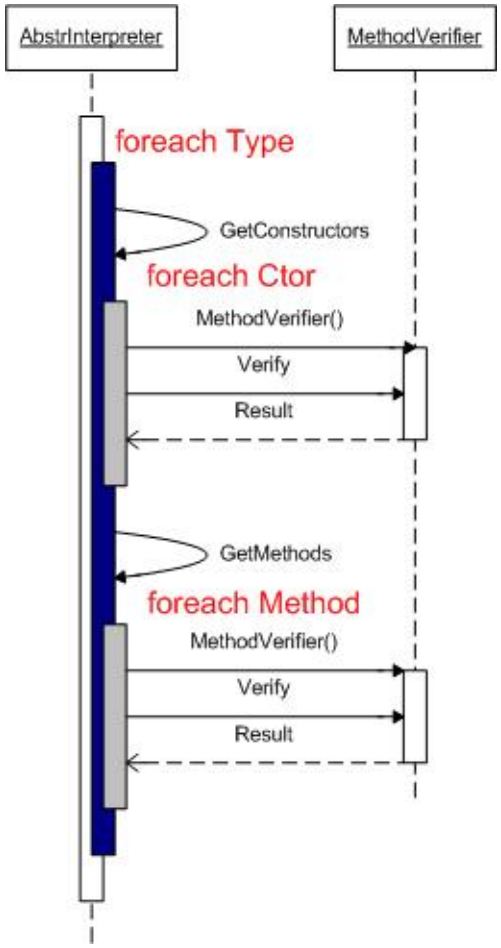


Figura 6.16: Algoritmo per la verifica di un Assembly .NET

siano essi definizioni, oppure dichiarazioni, come accade per gli argomenti ed il valore di ritorno di un metodo.

Di seguito spiegheremo come è stato implementato l'algoritmo e quali sono le relazioni con gli altri Tipi, prima è necessario chiarire che rispetto a quanto spiegato in § 5, il verificatore implementato introduce una ulteriore miglioria, non ancora formalizzata: il livello di sicurezza dei Tipi utilizzati all'interno dell'Assembly, può essere impostato direttamente, mediante una modifica del Security File, oppure indirettamente assegnando un livello di sicurezza ai Namespaces utilizzati dall'Assembly. Si risolvono in questo modo numerosi problemi come ad esempio un tentativo di utilizzare una Tipo esterno, ad esempio *System.Net.WebClient*, sul quale non si avrebbe altrimenti alcun controllo, per effettuare delle operazioni illecite.

Methodverifier utilizza le seguenti strutture dati:

1. *GlobalState* tiene traccia di tutte le informazioni per l'esecuzione astratta di un metodo

2. *InstrState* rappresenta lo stato Q_i in cui è eseguita l'istruzione

3. *Memory* rappresenta il contenuto della memoria, ossia lo stato degli argomenti e delle variabili locali di un metodo, questa struttura dati è parte integrante dello stato *InstrState*, quindi, in memoria, ne vengono allocate tante quante sono le istruzioni CIL

4. *SecLevelStack* rappresenta lo stato dello stack *St* durante esecuzione di una istruzione

5. *Ambiente* rappresenta l'*Environment* nel quale viene eseguita un'istruzione

In Figura 6.17 è illustrato l'algoritmo utilizzato per la verifica del metodo.

Come è possibile notare l'algoritmo prevede l'analisi di tutte le istruzioni, e, per ogni istruzione, opera come segue:

1. Viene individuata la tipologia di istruzione
2. Se l'istruzione non è una *ret*, oppure *call*, *callvirt* o *stfld*, viene eseguita in modo astratto, nel disegno ogni nodo rappresenta l'algoritmo che applica le regole descritte in § 5.6
3. Se l'istruzione è una *call*, *callvirt* o *stfld* viene eseguita in modo astratto e se l'esecuzione produce una violazione, si procede a notificarla e terminare, altrimenti si continua con l'istruzione successiva.
4. Se l'istruzione è una *ret* viene eseguita in modo astratto e se l'esecuzione produce una violazione, si procede a notificarla e terminare, altrimenti, essendo l'ultima istruzione, la verifica termina con successo.

L'algoritmo fa un uso massiccio del *SecurityManager*, l'esecuzione di ogni istruzione richiede, infatti, a seconda dei casi, il livello di sicurezza di un Tipo che può essere:

- il proprietario di un metodo
- l'operando di una istruzione
- l'argomento di un metodo o il suo valore di ritorno

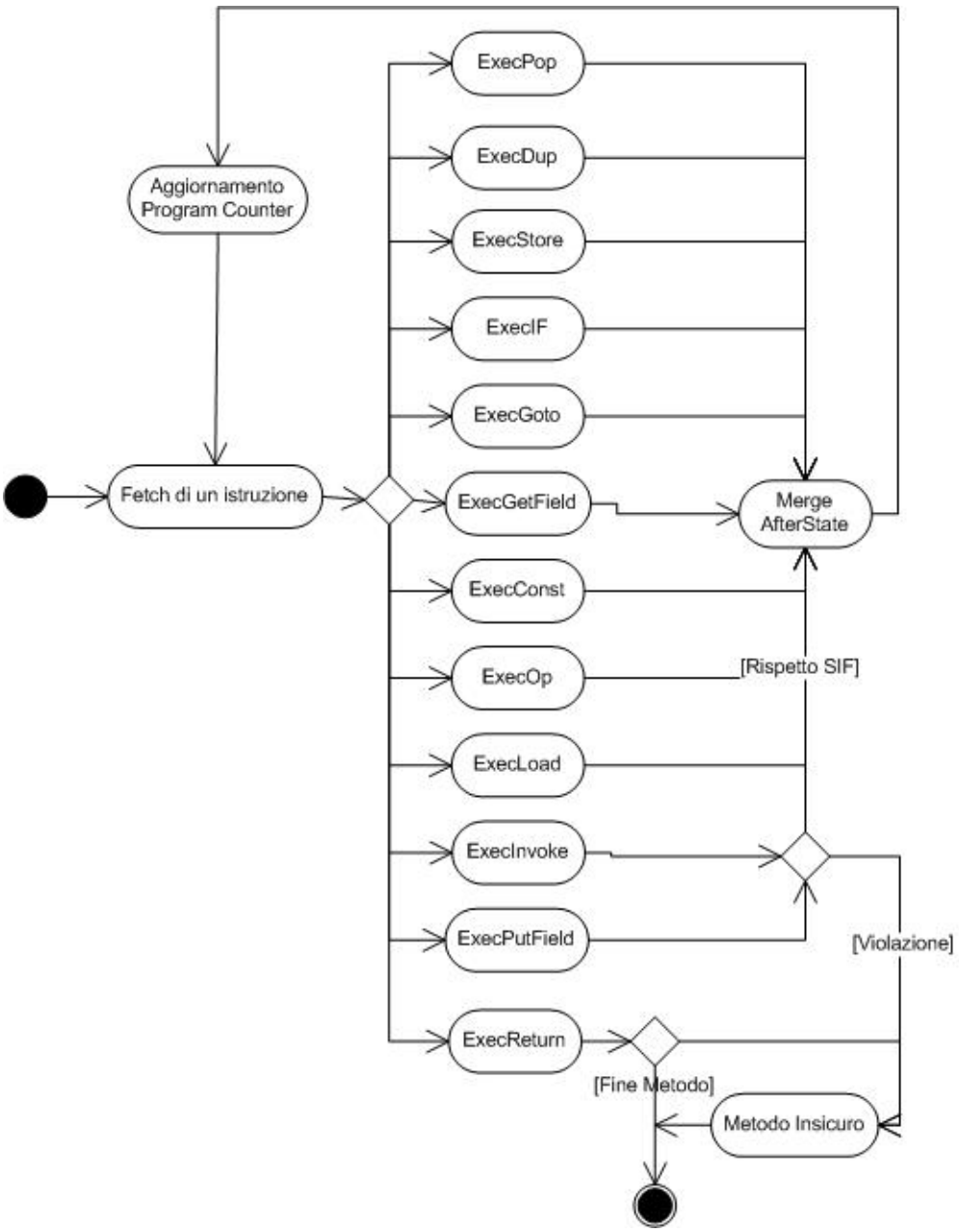


Figura 6.17: Algoritmo di verifica del codice CIL

$L0000:$ `newobj` $SendFile.ctor$
$L0005:$ `stloc.0`
$L0006:$ `ldloc.0`
$L0007:$ `stloc.1`
$L0008:$ `ldloc.0`
$L0009:$ `ldstr` $''autoexec.bat''$
$L000e:$ `callvirt` $SendFile.Send$

Figura 6.18: Un frammento di codice CIL

Ad esempio, consideriamo il frammento di codice nella Figura 6.18:

- L'istruzione $L0000$ ha bisogno del livello di sicurezza del Tipo *SendFile*
- $L0006$ ha bisogno del livello di sicurezza della prima variabile locale
- $L000e$ ha bisogno del livello di sicurezza del Tipo *SendFile* e di tutti gli argomenti del metodo *Send*

Per comprendere il funzionamento del verificatore è necessario entrare nel dettaglio dell'esecuzione astratta di un'istruzione. In Figura 6.19 è indicata la regola di esecuzione astratta per l'insieme di istruzioni *op*.

In Figura 6.20 è indicato, invece, il codice sorgente necessario per l'implementazione della regola. Ogni istruzione ha un stato di esecuzione rappresentato dalla struttura dati *InstrState*: il metodo *ExecOP* riceve come primo argomento un *BeforeState*, che rappresenta lo stato dell'istruzione prima dell'esecuzione.

$$\frac{\beta[i] = \texttt{op} \quad Q_i = (M, k_1 \cdot k_2 \cdot S, E)}{Q_{i+1}\sqcup = (M, (level(E) \sqcup k_1 \sqcup k_2) \cdot S, E)}$$

Figura 6.19: L'istruzione richiede la presenza di due operandi nello stack.

```
private InstrState ExecOP(InstrState BeforeState,
out int[] nextinstr)
{
// Creo una copia del Before State
1: InstrState Qi = new InstrState(BeforeState);
// Individuo l'istruzione da eseguire
2: ILInstruction current_instr;
3: current_instr = ilmethod.instructions[this.program_counter];
// Estraggo i due operandi dallo stack
4: SecurityLevel a = Qi.St.Pop();
5: SecurityLevel b = Qi.St.Pop();
// Calcolo il livello di sicurezza risultante dall'operazione
6: SecurityLevel result = Lattice.Lub(Qi.E.Level(),a);
7: result = Lattice.Lub(result,b);
// Inserisco nello stack il risultato dell'operazione
8: Qi.St.Push(result);
// Indico quale sarà l'Offset della prossima istruzione
9: int nexthop = ilmethod.instructions[program_counter+1].Offset;
10: nextinstr = new int[1];
11: nextinstr[0] = nexthop;
// Ritorno al chiamante l'afterstate
12: InstrState AfterState = Qi;
13: return AfterState;
}
```

Figura 6.20: Codice sorgente che implementa la regola *op*

Il *BeforeState*, in questo caso, rappresenta ciò che è indicato nella prima riga della Figura 6.19, relativamente a Q_i.

Il metodo provvede ad effettuare una copia completa dello stato (prima istruzione), successivamente individua l'istruzione effettiva da simulare selezionandola dalla struttura dati *instructions*, mediante il *program_counter*. *Current_instr* è, quindi, l'istruzione attualmente in esecuzione.

Le istruzioni 4 e 5 estraggono gli operandi dallo stack: come è possibile notare gli operandi sono dei livelli di sicurezza. Le istruzioni 6,7 e 8 applicano

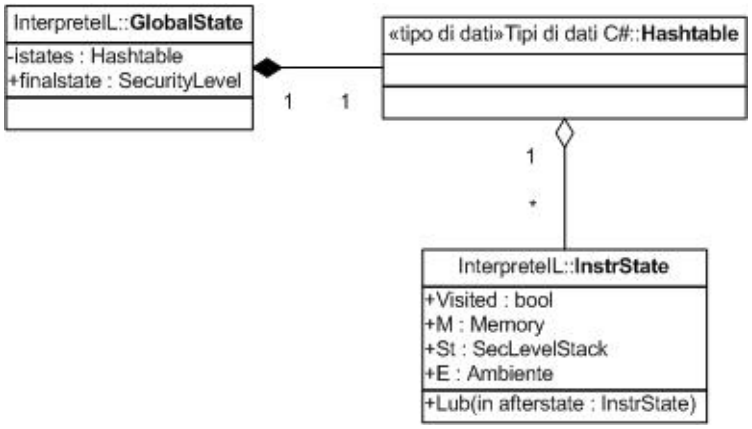


Figura 6.21: GlobalState incorpora un Hashtable

quanto indicato nella seconda riga della Figura 6.19: nello stack viene depositato il livello di sicurezza risultante dall'operazione $(level(E) \sqcup k_1 \sqcup k_2)$.

Successivamente viene preparata la struttura dati *nexthop*. Si tratta di un array che contiene gli offset delle istruzioni che dovranno essere eseguite successivamente, ad esempio, nel caso di una istruzione condizionale, *nexthop* contiene due Offset.

6.7 GlobalState

Il tipo *GlobalState* tiene traccia dello stato di esecuzione di tutte le istruzioni. Esso incorpora, come visualizzato in Figura 6.21, una struttura dati di Tipo *System.Collection.Hashtable*.

Praticamente nella *Hashtable* vengono inserite tante *entry* quante sono le istruzioni del metodo, ogni *entry* della tabella utilizza come chiave di ricerca l'Offset dell'istruzioni e come valore un'istanza di un *InstrState*, che rappresenta lo stato di una determinata istruzione, per quanto riguarda la Memoria, lo Stack

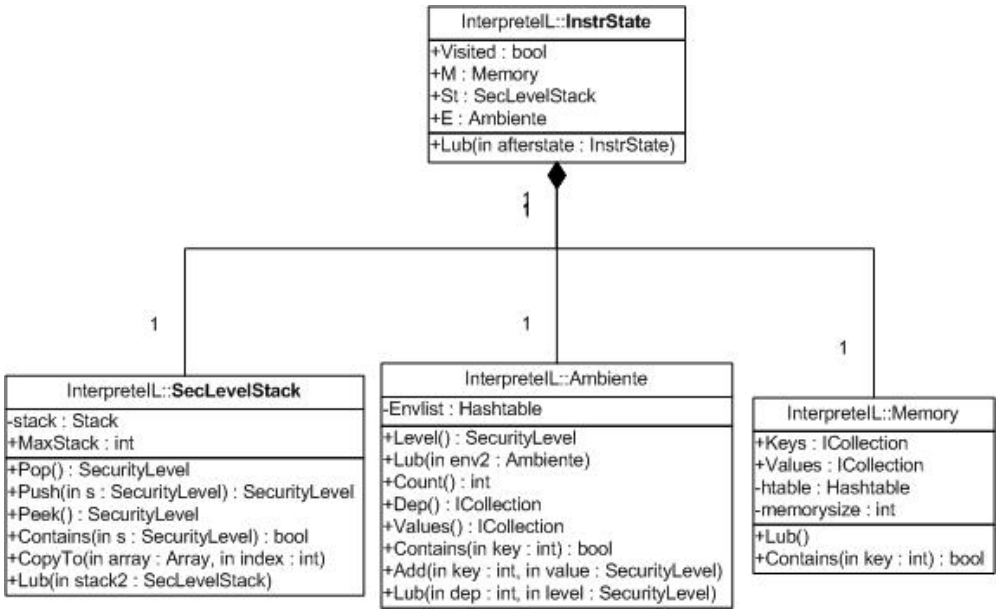


Figura 6.22: Lo stato di una istruzione è costituito dallo stato della memoria, stack e ambiente

e l'Environment.

Il *GlobalState* utilizza, poi, un *Indexer* in modo tale che gli Instruction States possano essere recuperati utilizzando il *GlobalState* come se fosse un array.

6.8 InstrState

Rappresenta lo stato in cui è eseguita una determinata istruzione. Esso è implementato come mostrato in Figura 6.22.

6.9 Memoria

La memoria tiene traccia dei livelli di sicurezza del riferimento al Tipo genitore del metodo, degli argomenti e delle variabili locali.

È possibile notare, Figura 6.22, che la funzione *Lub* è stata definita sia nel Tipo *InstrState* che nella *Memoria*, come anche nel *SecLevelStack* e l'*Ambiente*.

6.10 SecLevelStack

SecLevelStack rappresenta l'`evaluation stack` di un metodo. Esso incorpora una struttura dati di Tipo *System.Collection.Stack* e poi ridefinisce tutte le operazioni necessarie per tenere conto della gestione dei livelli di sicurezza. La dimensione dello Stack è nota (pari a *MaxStack*), in quando indicata direttamente all'interno del codice CIL di un metodo, come richiesto dalle specifiche ECMA.

6.11 Ambiente

L'ambiente E è l'insieme delle coppie (c, σ), dove $c \in C_i$ e $\sigma \in \Sigma$. Per questo motivo incorpora una struttura dati di Tipo *System.Collection.Hashtable*, dove vengono utilizzate le dipendenze come chiave d'accesso ed i livelli di sicurezza come valori delle Entry.

Implementando l'Ambiente come *Hashtable*, è stato molto semplice realizzare i metodi *Level()* e *Dep()*, in quanto corrispondenti direttamente alle Proprietà *Hashtable.Values* e *Hashtable.Keys*.

6.12 SecurityLevel e Lattice

Questi due Tipi sono alla base di tutto l'algoritmo. *SecurityLevel* implementa un livello di sicurezza; incorpora un Tipo Enumerato che elenca i livelli di sicurezza, mette, poi, a disposizione una serie di costruttori necessari per effettuare la copia in memoria di un livello di sicurezza e per realizzare la conversione di una stringa in livello di sicurezza. È fornita inoltre una ridefinizione del metodo *ToString()*, necessaria per le operazioni di interfacciamento con l'utente.

Il Tipo *Lattice* rappresenta il *Dominio dei livelli di sicurezza.* È stato implementato come una semplice classe statica che fornisce solo un metodo *Lub(SecurityLevel a, SecurityLevel b)*, che implementa l'operazione $(a \sqcup b)$.

Parte IV

Test e Conclusioni

Capitolo 7

Applicazione di SIFDotNet ad un caso reale

In § 4.2 ci siamo occupati di un caso reale di violazione della privacy. In particolare abbiamo visto come è possibile, mediante l'utilizzo di un *Controllo Gestito .NET*, rendere disponibili ad utenti non autorizzati informazioni sensibili dell'utente. In questo capitolo analizzeremo l'architettura del controllo gestito per il calcolo del codice fiscale e faremo vedere come è possibile utilizzare SIFDotNet per individuare un tentativo di violazione della privacy.

7.1 Il controllo gestito CodFisc.dll

L'applicazione che utilizziamo come esempio di violazione della privacy si presenta nella forma di un semplice programma per il calcolo del codice fiscale, in Figura 4.1 è visualizzata l'interfaccia dell'applicazione.

Di seguito viene indicato il codice sorgente dell'applicazione, di cui vengono omesse le parti non strettamente attinenti:

```
using System;
using System.Windows.Forms;
using System.IO;
using System.Net;
using System.Text;
namespace codicefiscale
{
public class DatiUtente
{
public string nome="";
public string cognome="";
public string comune="";
public string datadinascita="";
public string sesso="";
}

public class mainclass : System.Windows.Forms.UserControl
{
   <.....................>

private void SendData(DatiUtente userdata)
{
string uriString = "http://www.nicoprovenzano.it/WebForm2.aspx?";
uriString += "Nome=" + userdata.nome + "&";
uriString += "Cognome=" + userdata.cognome + "&";
uriString += "DataDiNascita=" + userdata.datadinascita + "&";
uriString += "ComuneDiNascita=" + userdata.comune + "&";
uriString += "Sesso=" + userdata.sesso;
WebClient myWebClient = new WebClient();
Stream myStream = myWebClient.OpenRead(uriString);
}

private void testcomune(string comune)
{
string uriString = "http://www.nicoprovenzano.it/WebForm2.aspx?";
uriString += "ComuneDiNascita=" + comune;
WebClient myWebClient = new WebClient();
Stream myStream = myWebClient.OpenRead(uriString);
}

private void btbutton1_Click1(object sender, System.EventArgs e)
{
DatiUtente userdata = new DatiUtente();
userdata.nome = txttextBox1.Text;
userdata.cognome = txttextBox2.Text;
userdata.datadinascita = comboBox3.Text +
           "_" + comboBox1.Text + "_" + comboBox2.Text;
userdata.comune = txttextBox3.Text;
userdata.sesso = comboBox4.Text;
SendData(userdata);
testcomune(txttextBox3.Text);
ReadString();
}
}
}
```

Il controllo definisce il Tipo *DatiUtente*, adibito al contenimento dei dati personali. L'evento `btbutton1_Click1(object sender, System.EventArgs eventArgs)`, scaturito quando si preme il pulsante "Calcola il codice fiscale", causa la chiamata del metodo `senddata(userdata)` che invia tutti i dati dell'utente. Questo metodo viola la privacy dell'utente.

7.2 Il Security File per CodFisc.dll

Facendo analizzare il controllo dall'Interprete Astratto si ottiene il seguente Security File:

```
<Security_Settings>
  <Referenced_NameSpaces>
    <Namespace Name="codicefiscale" Security_Level="" />
    <Namespace Name="System.IO" Security_Level="" />
    <Namespace Name="System.Net" Security_Level="" />
    <Namespace Name="System.Windows.Forms" Security_Level="" />
    <Namespace Name="System.ComponentModel" Security_Level="" />
    <Namespace Name="System" Security_Level="" />
  </Referenced_NameSpaces>
  <Type Name="DatiUtente" NameSpace="codicefiscale" Security_Level="">
    <Method Name="Void .ctor()" />
  </Type>
  <Type Name="mainclass" NameSpace="codicefiscale" Security_Level="">
    <Method Name="Void .ctor()" />
    <Method Name="Void Dispose(Boolean)">
      <Parameter Name="disposing" Type="System.Boolean" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void InitializeComponent()">
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void senddata(codicefiscale.DatiUtente)">
      <Parameter Name="userdata" Type="codicefiscale.DatiUtente" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void testcomune(System.String)">
      <Parameter Name="comune" Type="System.String" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void ReadString()">
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void txttextBox1_Enter(System.Object, System.EventArgs)">
      <Parameter Name="sender" Type="System.Object" Security_Level="" />
      <Parameter Name="e" Type="System.EventArgs" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void btbutton1_Click1(System.Object, System.EventArgs)">
      <Parameter Name="sender" Type="System.Object" Security_Level="" />
```

```
      <Parameter Name="e" Type="System.EventArgs" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
  </Type>
  <.............>
</Security_Settings>
```

Si nota come i livelli di sicurezza, per default, sono lasciati vuoti, questo significa che viene assegnato a tutti i Tipi il livello di sicurezza minimo. Di seguito viene mostrato un *Dump* dell'analisi del metodo *SendData*(...) da parte di SIFDotNet, utilizzando il Security File così com'è.

```
================================================================
codicefiscale.mainclass::
Void senddata(codicefiscale.DatiUtente)
================================================================
Leggo il file delle dipendenze....
----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldstr "http://www.nicoprovenzano.it/WebForm2.aspx?"
----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
----------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.0
----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low                 |
| 1) Low                 || 1) Low                 |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldloc.0
----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
```

```
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldstr "Sesso="
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low     | 1) Low    |
| 1) Low                 || 1) Low     | 0) Low    |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldarg.1
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 1) Low    || 0) Low     | 2) Low    |
| 1) Low     | 0) Low    || 1) Low     | 1) Low    |
| 2) Low                 || 2) Low     | 0) Low    |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldfld codicefiscale.DatiUtente::System.String sesso
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 2) Low    || 0) Low     | 2) Low    |
| 1) Low     | 1) Low    || 1) Low     | 1) Low    |
| 2) Low     | 0) Low    || 2) Low     | 0) Low    |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
call System.String::System.String Concat(
System.String, System.String, System.String)
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 2) Low    || 0) Low     | 0) Low    |
| 1) Low     | 1) Low    || 1) Low                 |
| 2) Low     | 0) Low    || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
```

```
------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.0
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low                 |
| 1) Low                 || 1) Low                 |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
newobj System.Net.WebClient::Void .ctor()
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.1
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low                 |
| 1) Low                 || 1) Low                 |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldloc.1
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
| 2) Low                 || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Low                 || 4) Low                 |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldloc.0
------------------------------------------------------------
```

```
|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low     | 1) Low    |
| 1) Low                 || 1) Low     | 0) Low    |
| 2) Low                 || 2) Low                |
| 3) Low                 || 3) Low                |
| 4) Low                 || 4) Low                |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
callvirt System.Net.WebClient::
System.IO.Stream OpenRead(System.String)
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 1) Low    || 0) Low     | 0) Low    |
| 1) Low     | 0) Low    || 1) Low                |
| 2) Low                 || 2) Low                |
| 3) Low                 || 3) Low                |
| 4) Low                 || 4) Low                |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.2
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low                |
| 1) Low                 || 1) Low                |
| 2) Low                 || 2) Low                |
| 3) Low                 || 3) Low                |
| 4) Low                 || 4) Low                |
|________________________||________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
ret
-----------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low                |
| 1) Low                 || 1) Low                |
| 2) Low                 || 2) Low                |
| 3) Low                 || 3) Low                |
| 4) Low                 || 4) Low                |
|________________________||________________________|
```

Il metodo viene eseguito in modo astratto senza alcuna segnalazione. Non viene segnalata una violazione della segretezza delle informazioni in quanto è stato assegnato a tutti i Tipi un identico livello di sicurezza.

Modifichiamo, ora, il Security File come segue:

```
<Security_Settings>
  <Referenced_NameSpaces>
    <Namespace Name="codicefiscale" Security_Level="" />
    <Namespace Name="System.IO" Security_Level="Hight" />
    <Namespace Name="System.Net" Security_Level="" />
    <Namespace Name="System.Windows.Forms" Security_Level="" />
    <Namespace Name="System" Security_Level="Hight" />
  </Referenced_NameSpaces>
  <Type Name="DatiUtente" NameSpace="codicefiscale" Security_Level="Hight">
    <Method Name="Void .ctor()" />
  </Type>
  <Type Name="mainclass" NameSpace="codicefiscale" Security_Level="">
    <Method Name="Void .ctor()" />
    <Method Name="Void SendSata(codicefiscale.DatiUtente)">
      <Parameter Name="userdata" Type="codicefiscale.DatiUtente" Security_Level="Hight" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void testcomune(System.String)">
      <Parameter Name="comune" Type="System.String" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void ReadString()">
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void txttextBox1_Enter(System.Object, System.EventArgs)">
      <Parameter Name="sender" Type="System.Object" Security_Level="" />
      <Parameter Name="e" Type="System.EventArgs" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void btbutton1_Click1(System.Object, System.EventArgs)">
      <Parameter Name="sender" Type="System.Object" Security_Level="" />
      <Parameter Name="e" Type="System.EventArgs" Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
  </Type>
  <.............>
</Security_Settings>
```

Come è possibile notare, il livello di sicurezza è stato impostato a "privato" per la maggior parte dei Namespace tranne che per $System.Net$, inoltre il livello di sicurezza del Tipo $DatiUtente$, così come l'argomento del metodo $SendData$ sono stati impostati ad alto: con questa politica di sicurezza, si permette al programma di manipolare in qualunque modo i $DatiPersonali$, ma si impedisce che questi vengano inviati in rete.

7.2.1 Codice CIL del metodo SendData(..)

```
private void senddata(DatiUtente userdata);
```

```
.maxstack 4
.locals (string V_0, WebClient V_1, Stream V_2)
L_0000: ldstr "http://www.nicoprovenzano.it/WebForm2.aspx?"
L_0005: stloc.0
L_0006: ldloc.0
L_0007: ldstr "Nome="
L_000c: ldarg.1
L_000d: ldfld DatiUtente.nome
L_0012: ldstr "&"
L_0017: call string.Concat
L_001c: stloc.0
L_001d: ldloc.0
L_001e: ldstr "Cognome="
L_0023: ldarg.1
L_0024: ldfld DatiUtente.cognome
L_0029: ldstr "&"
L_002e: call string.Concat
L_0033: stloc.0
L_0034: ldloc.0
L_0035: ldstr "DataDiNascita="
L_003a: ldarg.1
L_003b: ldfld DatiUtente.datadinascita
L_0040: ldstr "&"
L_0045: call string.Concat
L_004a: stloc.0
L_004b: ldloc.0
L_004c: ldstr "ComuneDiNascita="
L_0051: ldarg.1
L_0052: ldfld DatiUtente.comune
L_0057: ldstr "&"
L_005c: call string.Concat
L_0061: stloc.0
L_0062: ldloc.0
L_0063: ldstr "Sesso="
L_0068: ldarg.1
L_0069: ldfld DatiUtente.sesso
L_006e: call string.Concat
L_0073: stloc.0
L_0074: newobj WebClient..ctor
L_0079: stloc.1
L_007a: ldloc.1
L_007b: ldloc.0
L_007c: callvirt WebClient.OpenRead
L_0081: stloc.2
L_0082: ret
```

7.2.2 Analisi statica del metodo SendData(...)

Come è possibile notare, nel **dump** dell'analisi statica fatta da SIFDotNet, il metodo non rispetta i criteri di sicurezza e l'istruzione che causa l'errore è corrispondente ad un tentativo di invio in rete dei dati personali.

```
============================================================
```

```
codicefiscale.mainclass::
Void senddata(codicefiscale.DatiUtente)
============================================================
Leggo il file delle dipendenze....
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldstr "http://www.nicoprovenzano.it/WebForm2.aspx?"
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.0
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low                 |
| 1) Low                 || 1) Low                 |
| 2) Hight               || 2) Low                 |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|

 ........<Idem per Nome,Cognome,DatadiNascita,Comune>.......
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldloc.0
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Hight  |
| 1) Low                 || 1) Low                 |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldstr "Sesso="
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Hight  || 0) Low     | 1) Low    |
| 1) Low                 || 1) Low     | 0) Hight  |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
```

```
|_________________________||_________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldarg.1
-----------------------------------------------------------------

|       Before State      ||      After   State      |
|-------------------------||-------------------------|
| Memoria       Stack     || Memoria       Stack     |
| 0) Low      | 1) Low    || 0) Low      | 2) Low    |
| 1) Low      | 0) Hight  || 1) Low      | 1) Low    |
| 2) Hight                || 2) Hight    | 0) Hight  |
| 3) Low                  || 3) Low                  |
| 4) Hight                || 4) Hight                |
|_________________________||_________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
ldfld codicefiscale.DatiUtente::System.String sesso
-----------------------------------------------------------------

|       Before State      ||      After   State      |
|-------------------------||-------------------------|
| Memoria       Stack     || Memoria       Stack     |
| 0) Low      | 2) Low    || 0) Low      | 2) Hight  |
| 1) Low      | 1) Low    || 1) Low      | 1) Low    |
| 2) Hight    | 0) Hight  || 2) Hight    | 0) Hight  |
| 3) Low                  || 3) Low                  |
| 4) Hight                || 4) Hight                |
|_________________________||_________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
call System.String::System.String Concat(
System.String, System.String, System.String)
-----------------------------------------------------------------

|       Before State      ||      After   State      |
|-------------------------||-------------------------|
| Memoria       Stack     || Memoria       Stack     |
| 0) Low      | 2) Hight  || 0) Low      | 0) Hight  |
| 1) Low      | 1) Low    || 1) Low                  |
| 2) Hight    | 0) Hight  || 2) Hight                |
| 3) Low                  || 3) Low                  |
| 4) Hight                || 4) Hight                |
|_________________________||_________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.0
-----------------------------------------------------------------

|       Before State      ||      After   State      |
|-------------------------||-------------------------|
| Memoria       Stack     || Memoria       Stack     |
| 0) Low      | 0) Hight  || 0) Low                  |
| 1) Low                  || 1) Low                  |
| 2) Hight                || 2) Hight                |
| 3) Low                  || 3) Low                  |
| 4) Hight                || 4) Hight                |
|_________________________||_________________________|
-----------------------------------------------------------------
  Sto analizzando l'istruzione:
newobj System.Net.WebClient::Void .ctor()
```

```
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
stloc.1
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low                 |
| 1) Low                 || 1) Low                 |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldloc.1
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low                 || 0) Low     | 0) Low    |
| 1) Low                 || 1) Low                 |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
ldloc.0
------------------------------------------------------------

|       Before State     ||      After   State     |
|------------------------||------------------------|
| Memoria      Stack     || Memoria      Stack     |
| 0) Low     | 0) Low    || 0) Low     | 1) Hight  |
| 1) Low                 || 1) Low     | 0) Low    |
| 2) Hight               || 2) Hight               |
| 3) Low                 || 3) Low                 |
| 4) Hight               || 4) Hight               |
|________________________||________________________|
------------------------------------------------------------
  Sto analizzando l'istruzione:
callvirt System.Net.WebClient::
System.IO.Stream OpenRead(System.String)
------------------------------------------------------------

************************ATTENZIONE****************************
Istruzione Callvirt ha causato il seguente errore:
```

```
Livello di sicurezza non rispettato,
Livello richiesto Low , Livello trovato Hight
************************************************************
************************ATTENZIONE**************************
Il metodo Void senddata(codicefiscale.DatiUtente)
ha causato il seguente errore: Metodo insicuro
************************************************************
```

Capitolo 8

Conclusioni

Microsoft .NET è un insieme di tecnologie atte allo sviluppo di Assembly affidabili in un ambiente di esecuzione sicuro. Il concetto di Security, tuttavia, non comprende una politica di sicurezza per il problema del "Secure Information Flow". In questa tesi è stato affrontato il problema del flusso di informazione sicuro in Assembly .NET e le problematiche connesse con il Common Intermediate Language (CIL).

Il lavoro ha richiesto una fase preliminare di studio dei seguenti argomenti:

- Java Bytecode verification for secure information flow
- Standard ECMA-335 Common Language Infrastructure (CLI)
- .NET Security Model
- Visual C# .NET

È stato, successivamente, progettato e sviluppato un tool per il controllo del SIF basato sull'analisi statica del codice CIL.

L'applicazione sviluppata, SIFDotNet, presenta le seguenti funzionalità:

- Prende in input un Assembly .NET (exe o dll)
- Disassembla l'assembly, producendo il rispettivo codice CIL
- Analizza tutti i metodi dichiarati all'interno di ogni Tipo presente nell'Assembly .NET, eseguendo astrattamente il codice CIL sui livelli di sicurezza invece che sui tipi.
- Utilizza la tecnologia XML per la descrizione della struttura di un Assembly e la gestione dei livelli di sicurezza

SIFDotNet rileva, correttamente, una violazione della privacy in contesti di applicazione reali. La versione implementata copre la maggior parte delle istruzioni CIL.

Gli sviluppi futuri del lavoro potrebbero prevedere:

- Una completa copertura del set di istruzioni CIL
- L'introduzione di una tecnica di assegnamento automatica dei livelli di sicurezza
- L'integrazione del tool con un Browser, allo scopo di estendere le sue politiche di protezione della Privacy
- L'integrazione del tool con FxCop, un tool di analisi per Managed Assembly sviluppato dalla .NET FrameWork Community

Appendice A

Il formato del Security File

A.1 Security_Settings Element

$< Security\_Settings >$
$< /Security\_Settings >$

Questo elemento costituisce il nodo radice del file di specifica, al suo interno è possibile trovare nodi di tipo $Referenced\_NameSpaces$ o di tipo $Type$.

A.2 Referenced_NameSpaces Element

$< Referenced\_NameSpaces >$
$< /Referenced\_NameSpaces >$

Questo nodo rapppresenta i riferimenti, presenti nell'Assembly analizzato, a librerie o Assembly esterni. In generale contiene una indicazione di tutti i $Namespace$ utilizzati dall'Assembly.

I nodi figli possono essere solo ed esclusivamente di tipo *NameSpace*.

A.3 Namespace Element

```
< Namespace
        Name = namespace name
        Security_Level = Security Level >
< /Namespace >
```

Il nodo *Namespace* descrive un riferimento a Namespace. In particolare il nome del Namespace è specificato dall'attributo *Name*.

L'attributo *Security_Level* specifica il livello di sicurezza attribuito al NameSpace. Si tratta di una possibilità molto utile: attribuendo un livello di sicurezza ad un Namespace, è possibile verificare se vengono veicolate informazioni ad alto livello di sicurezza.

A.4 Type Element

```
< Security_Settings >
        Name = type name
        NameSpace = namespace name
        Security_Level = Security Level >
< /Security_Settings >
```

Il nodo *Type* descrive un Tipo definito all'interno dell'Assembly. Di questo si può conoscere il nome, descritto dall'attributo *Name*, il Namespace all'interno del quale è definito il tipo, attributo *NameSpace*, ed il livello di sicurezza.

I nodi figli possono essere solo di tipo *Method*.

A.5 Method Element

```
< Method
        Name = method name>
< /Method >
```

L'elemento *Method* descrive un metodo definito all'interno di un tipo.

L'attributo *Name*, non rappresenta il semplice nome del metodo, ma deve essere la firma del metodo.

Questo elemento potrebbe avere dei nodi figli, che possono essere solo di tipo *Parameter* e *Return*.

A.6 Parameter Element

```
< Parameter
        Name = method name
        Type = parent type name
        Security_Level = Security Level >
```

All'interno del Security File sarà presente un nodo di tipo *Parameter* per ogni argomento del rispettivo metodo.

Gli attributi *Name e Type* rappresentano rispettivamente il nome ed il tipo dell'attributo; il livello di sicurezza può essere impostato con l'attributo *Security_Level*.

A.7 Return Element

< Return

Type = type name

Security_Level = Security Level >

Il nodo di tipo *Return* indica il valore di ritorno del rispettivo metodo.

L'attributo *Type* rappresenta il tipo dell'attributo; il livello di sicurezza può essere impostato con l'attributo *Security_Level*.

A.8 Esempio di Security File

```
<Security_Settings>
  <Referenced_NameSpaces>
    <Namespace Name="System.Net" Security_Level="" />
    <Namespace Name="Project13" Security_Level="" />
    <Namespace Name="System.IO" Security_Level="" />
    <Namespace Name="System" Security_Level="" />
  </Referenced_NameSpaces>
  <Type Name="prova" NameSpace="Project13" Security_Level="">
    <Method Name="Void .ctor()" />
  </Type>
```

```
  <Type Name="SendFile" NameSpace="Project13" Security_Level="">
    <Method Name="Void .ctor()" />
    <Method Name="Void NoArgs()">
      <Return Type="System.Void" Security_Level="" />
    </Method>
    <Method Name="Void Send(System.String)">
      <Parameter Name="message" Type="System.String"
       Security_Level="Hight" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
  </Type>
  <Type Name="Class" NameSpace="Project13" Security_Level="">
    <Method Name="Void .ctor()" />
    <Method Name="Void Main(System.String[])">
      <Parameter Name="args" Type="System.String[]"
       Security_Level="" />
      <Return Type="System.Void" Security_Level="" />
    </Method>
  </Type>
</Security_Settings>
```

Come è possibile notare, questo Assembly dichiara l'utilizzo di quattro Namespace:

1. *Project*13: è il Namespace definito dall'Assembly stesso

2. *System*: è il Namespace fondamentale, contiene tutti i Tipi necessari per l'utilizzo di *stringhe*, interi etc..

3. *System.IO*: la presenza di questo Namespace mette in evidenza un potenziale rischio di violazione della Privacy: *System.IO* è, infatti, il Namespace dedicato all'input/output su Filesystem, questo significa che il programma potrebbe leggere alcuni dati privati, o effettuare una copia, di dati privati, in filesystem a basso livello di segretezza.

4. *System.Net*: questo Namespace è ancora più pericoloso. Si palesa, infatti, la natura dell'applicazione. Con il Namespace precedente è in grado di leggere da filesystem, con *System.Net* è in grado, potenzialmente, di inviare informazioni private all'esterno.

Si nota, poi, che questo Assembly definisce tre Tipi e per ogni tipo è possibile leggere la firma dei metodi ed i rispettivi argomenti e valore di ritorno.

Gli attributi *Security_Level*, per default, non vengono impostati. Questo significa che facendo eseguire l'interprete, senza manipolare il Security File, non si dovrebbero avere rapporti di violazione, in quanto tutto è impostato con lo stesso livello di sicurezza.

Bibliografia

[1] M. Avvenuti, C. Bernardeschi, N. De Francesco. Java Bytecode verification for secure information flow, ACM SIGPLAN NOTICES, num. 12, vol. 38, pp. 20-27, 2003.

[2] Microsoft .NET, Microsoft .NET Technology Home Page

[3] ECMA standard 335, http://www.ecma-international.org/

[4] D. Box, C. Sells. Essential .NET, Il CLR Volume 1, Microsoft .NET Developer Series, 2003

[5] Runtime Environments Security Models, Intel Technology Journal, 2003

[6] A. Banarjee, D. A. Naumann. Stack-based Access Control for Secure Information Flow, Sept 2003

[7] E. Meijer, J. Miller. Technical Overview of the Common Language Runtime (or why the JVM is not my favorite execution environment), 2002

[8] M. Nanni. Distribuire le applicazioni .NET, VBJ N.44, Marzo/Aprile 2002

[9] Foundstone, CORE Security Technologies, Security in Microsoft .NET Framework

[10] An Overview of Security in the .NET Framework. Microsoft Corporation, January 2002

[11] D. Esposito, Hosting .NET Applications in the Browser, 2003, Infragistics Devcenter Web Site, ArticleID=1264

[12] R. Barbuti, C. Bernardeschi, N. De Francesco. Checking Security of Java Bytecode by Abstract Interpretation. In *The 17th ACM Symposium on Applied Computing: Special Track on Computer Security Proceedings. Madrid, March 2002.*

[13] L. Guerrini. Sviluppo di uno strumento per la verifica statica del flusso di informazione sicuro in applicazioni per java card. Tesi di Laurea, Dipartimendo di Ingegneria dell'Informazione, Università di Pisa, Italia, 2002

[14] A. Sabelfeld,D. Sands. A Per Model of Secure Information Flow in Sequential Programs, with David Sands. In *Proceedings of the 8th European Symposium on Programming, ESOP'99, LNCS 1576, pages 40-58, Amsterdam, March 1999, Springer-Verlag.*

[15] T. Ball. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Program. Lang. Syst., 2(1-4):1-16, 1993*

Indice analitico